Talking bananas

Talking Bananas

Structural Recursion for Session Types

Sam Lindley J. Garrett Morris

The University of Edinburgh, UK

{Sam.Lindley,Garrett.Morris}@ed.ac.uk

Abstract

Session types provide static guarantees that concurrent programs respect communication protocols. We give a novel account of recursive session types in the context of GV, a small concurrent extension of the linear λ-calculus. We extend GV with recursive types and catamorphisms, following the initial algebra semantics of recursion, and show that doing so naturally gives rise to recursive session types. We show that this principled approach to recursion resolves long-standing problems in the treatment of duality for recursive session types.

We characterize the expressiveness of GV concurrency by giving a CPS translation to (non-concurrent) λ-calculus and proving that reduction in GV is simulated by full reduction in λ-calculus. This shows that GV remains terminating in the presence of positive recursive types, and that such arguments extend to other extensions of GV, such as polymorphism or non-linear types, by appeal to normalization results for sequential λ-calculi. We also show that GV remains deadlock free and deterministic in the presence of recursive types.

Finally, we extend CP, a session-typed process calculus based on linear logic, with recursive types, and show that doing so preserves the connection between reduction in GV and cut elimination in CP.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Concurrent, distributed, and parallel languages; D.3.3 [*Language Constructs and Features*]: Recursion

Keywords Session types, recursion

1. Introduction

Concurrency and communication have become central problems in modern software design and engineering, from hand-held applications relying on remote services to provide key functionality, through traditional applications now running on multi-core hardware, to distributed applications running across data centers. Assuring correct behavior for concurrent programs requires reasoning not just about the type of data communicated, but about the order in which communication takes place. For instance, the messages between an SMTP server and client are all strings representing SMTP commands, but a client that sends the recipient's address before the sender's address is in violation of the protocol despite having sent well-formed SMTP commands.

Session types, originally proposed by Honda [22], are an approach to statically verifying communicating concurrent programs. A session type specifies the expected communication along a channel. For example, consider a simplification of the client's view of the SMTP protocol. After being authenticated, a client has the option of sending one or more messages, each consisting of a sender's address, recipient's address, and message, in that order. We could express this with the following session type:

$$\mathit{Client} \stackrel{\text{def}}{=} !\mathit{FromAddress}.!\mathit{ToAddress}.!\mathit{Message}.\mathit{Client} \oplus^{!} !\mathit{Quit}.\mathsf{end}$$

where type constants *FromAddress*, *ToAddress*, *Message*, and *Quit* denote the corresponding SMTP commands. This definition makes use of several session type constructors. The type $!T.S$ denotes sending a value of type T before continuing with behavior S, $S \oplus^{!} S'$ denotes communicating a choice between behaviors S and S', end denotes the end of a session, and finally we make use of recursive definition to specify repetition in the protocol. A key aspect of session typing is duality. The session type of an SMTP server is dual to that of the client:

$$\mathit{Server} \stackrel{\text{def}}{=} ?\mathit{FromAddress}.?\mathit{ToAddress}.?\mathit{Message}.\mathit{Server} \oplus^{?} ?\mathit{Quit}.\mathsf{end}$$

This type definition uses dual features to those in the client's type: $?T.S$ denotes receiving a value of type T before continuing as S and $S \oplus^{?} S'$ denotes receiving a choice between S and S.

We present a novel account of recursive and corecursive session types. Following initial algebra semantics, we characterize recursive computation by catamorphisms (folds) rather than by an arbitrary fixed-point operator. Similarly, we characterize corecursive computation by anamorphisms (unfolds). This formulation differs from traditional presentations of recursive session types in three ways. First, we identify dual notions of recursion, corresponding to producers and consumers, rather than having a single self-dual notion of recursion in session types. Second, as they are based on well-founded recursive data types, our recursive session types guarantee termination and freedom from deadlock and livelock. Third, following algebraic ideas of recursion and duality leads to a sound syntactic characterization of duality for recursive session types. Many previous syntactic formulations of session type duality either incorrectly identify non-dual processes as dual [6, 7] or rely on corecursive expansion of session types [8].

We present our formulation of recursive and corecursive session types as an extension, called μGV [26], to a core concurrent λ-calculus called GV. We extend GV with (co)recursive types, (co)recursive session types, and (un)folds over both (co)recursive types and (co)recursive session types, and show that these are suf-

ficient to write non-trivial programs using (co)recursive session types. Previous work on GV has minimized the core calculus by encoding session-typed features in terms of functional features and simple input and output primitives. We continue this thread, showing that recursive session types can be encoded in terms of recursive data types. This result simplifies the concurrent semantics of μGV; for example, it allows us to apply previous results on deadlock freedom and determinism of GV to μGV unchanged. By identifying least and greatest fixed points in the resulting calculus, we obtain a system that admits non-terminating communication, but still guarantees deadlock freedom and productivity.

We also seek to characterize the expressiveness of μGV's concurrency. To do so, we give a CPS translation from GV into linear λ-calculus (without concurrency), and show that full reduction in the latter simulates reduction in the former. This also allows us to extend standard results on termination in sequential λ-calculi to results for termination of extensions of GV. Most immediately, we can conclude that μGV is terminating. The approach applies equally well to other extensions of GV, such as with polymorphism or non-linear types.

Recent work of Caires and Pfenning [13] and Wadler [34] develops a correspondence between reduction in process calculi and cut elimination in linear logic. GV is closely connected to Wadler's logic-based process calculus CP: in prior work [26] we give translations between GV and CP such that reduction in each simulates reduction in the other. We extend this observation to (co)recursive session types. We define an extension of CP to include (co)recursive types, following Baelde's formulation of fixed points in classical linear logic [3], and then extend the semantics-preserving translation between CP and GV to include recursive types.

Recent work by Toninho et al [32] explores corecursive session types from a propositions-as-types perspective. Despite having similar aims, our approach differs from theirs in three significant ways. First, we identify parallels between concurrent and sequential abstractions, in this case between recursive and corecursive data types and recursive session types. Toninho et al., in contrast, develop corecursive session types directly from the corresponding proof theory. This simplifies our concurrent semantics, as we do not have to account for recursive communication directly. Second, we identify two forms of recursive session types—corresponding to encodings based on recursive and corecursive data types—and that their composition can provide unbounded computation. Toninho et al. identify one of these forms, but not the other. Third, as illustrated by the equivalence with μCP, our session types are fundamentally classical, while Toninho et al. build on intuitionistic proof theory. Thus, for example, our results on the duality of recursive session types do not arise from their approach. We see the coincidence of our typing rules with theirs, despite the significant differences in methodology and foundations, as reinforcing the relevance of both lines of inquiry.

The paper proceeds as follows. We begin with our session-typed functional language, μGV. We present μGV's functional fragment (§2), extending a core linear λ-calculus with both recursive and corecursive types, and discuss the connection between μGV and languages with non-termination. We present the concurrent fragment of the language (§3), including both recursive and corecursive session types, and compare our approach to traditional characterizations of duality for session types. We give encodings for the concurrent features of μGV in terms of simpler constructs and give a concurrent small-step operational semantics for μGV (§4). We characterize the expressivity of GV concurrency by a CPS translation to a non-concurrent λ-calculus (§5). In doing so, we show that μGV is terminating, and thus (in combination with existing work on GV) free from livelock and deadlock. To establish a strong connection between μGV and linear logic, we present an extension of CP, called μCP, which includes least and greatest fixed points and corresponding recursive and corecursive proof terms, and show semantics-preserving translations from μGV to μCP and vice versa (§6). We conclude by discussing related (§7) and future (§8) work.

2. Functional μGV

We now describe Functional μGV, the functional fragment of μGV. We begin with Functional GV (§2.1), a linear λ-calculus without any form of recursion or iteration. We then discuss two extensions of GV. First, we extend Functional GV with recursive types and catamorphisms (§2.2). Second, we add corecursive types and anamorphisms (§2.3), and discusses the relationship with non-terminating recursion.

2.1 Functional GV

Functional GV is based on the multiplicative-additive fragment of intuitionistic linear logic. The syntax of Functional GV's terms and types is given at the top of Figure 1. Types include linear implication ($T \multimap U$), binary and nullary multiplicative products ($T \otimes U$ and $\mathbf{1}$), additive sums ($T \oplus U$ and $\mathbf{0}$), and additive products ($T \,\&\, U$ and $\top$). We write $M; N$ for $\mathsf{let}\ () = M\ \mathsf{in}\ N$. The syntax of terms includes (fully applied) constants $K\,M$, used to introduce concurrency; we give the constants and their typing rules in §3.

The typing rules for Functional GV are given in the center of Figure 1; most are standard for linear λ-calculus. The variable rule insists on a singleton environment. Rules for the multiplicative combinators ($T \multimap U$ and $T \otimes U$) split their hypotheses, while rules for the additive combinators ($T \,\&\, U$ and $T \oplus U$) duplicate them. In line with its interpretation as falsity, there is no introduction rule for $\mathbf{0}$, nor elimination rule for its dual $\top$.

A small-step operational semantics for Functional GV is given at the bottom of Figure 1. To maintain a close connection with cut-elimination, we define term reduction using weak explicit substitutions [25]. In this approach, we capture substitutions at λ-abstractions instead of applying them directly to the body of the abstraction. Our values thus include closures $\lambda^{\sigma}x.M$, which pair a function abstraction $\lambda x.M$ with a captured environment σ. We extend the typing judgment to include closures by

$$\frac{\Gamma, x : T \vdash M\sigma : U \quad dom(\sigma) = fv(M) \setminus \{x\}}{\Gamma \vdash \lambda^{\sigma}x.M : T \multimap U}$$

where we write $fv(M)$ to denote the free variables of M. The free variables of a closure are the free variables of the range of σ; capture-avoiding substitution $M\sigma$ is defined on the free variables of M. We implicitly treat plain abstractions $\lambda x.M$ as closures $\lambda^{\sigma}x.(M\sigma')$ where σ maps each free variable x_i of M to a fresh variable x_i', and σ' is the inverse of σ.

2.2 Recursion

Next, we extend Functional GV with recursive types and their terms. Our treatment of recursive types is based on the initial algebra semantics of recursion [21]. We extend our language with positive type functors F and their least fixed points μF. We introduce two new term forms: the operator in captures that μF is itself the carrier of an F-algebra, while the fold operator $(\!|-|\!)$ captures that μF is initial. This presentation of recursive types has a long history in the functional programming community, dating at least from the treatment of lists in Squiggol [9], and generalized by Meijer et al. [29].

The syntax and typing for recursive types are given at the top of Figure 2. We extend the language of types with type variables X and least fixed points μF, and terms as discussed before. We omit

Syntax

$$\begin{array}{lll} \text{Types} & T, U ::= & T \multimap U \mid T \otimes U \mid \mathbf{1} \mid T \oplus U \mid \mathbf{0} \\ & & \mid T \,\&\, U \mid \top \\ \text{Terms} & L, M, N ::= & x \mid K\,M \mid \lambda x.M \mid M\,N \\ & & \mid (M, N) \mid \mathsf{let}\ (x, y) = M\ \mathsf{in}\ N \\ & & \mid () \mid \mathsf{let}\ () = M\ \mathsf{in}\ N \\ & & \mid \mathsf{inl}\,M \mid \mathsf{inr}\,M \\ & & \mid \mathsf{case}\,L\,\{\mathsf{inl}\,x \mapsto M; \mathsf{inr}\,x \mapsto N\} \\ & & \mid \mathsf{absurd}\,M \\ & & \mid \langle M, N\rangle \mid \mathsf{fst}\,M \mid \mathsf{snd}\,M \mid \langle\rangle \end{array}$$

Typing

$$\frac{}{x : T \vdash x : T} \qquad \frac{K : T \multimap U \quad \Gamma \vdash M : T}{\Gamma \vdash K\,M : U}$$

$$\frac{\Gamma, x : T \vdash M : U}{\Gamma \vdash \lambda x.M : T \multimap U} \qquad \frac{\Gamma \vdash M : T \multimap U \quad \Gamma' \vdash N : T}{\Gamma, \Gamma' \vdash M\,N : U}$$

$$\frac{\Gamma \vdash M : T \quad \Gamma' \vdash N : U}{\Gamma, \Gamma' \vdash (M, N) : T \otimes U} \qquad \frac{\Gamma \vdash M : T \otimes T' \quad \Gamma', x : T, y : T' \vdash N : U}{\Gamma, \Gamma' \vdash \mathsf{let}\ (x, y) = M\ \mathsf{in}\ N : U}$$

$$\frac{}{\vdash () : \mathbf{1}} \qquad \frac{\Gamma \vdash M : \mathbf{1} \quad \Gamma' \vdash N : T}{\Gamma, \Gamma' \vdash \mathsf{let}\ () = M\ \mathsf{in}\ N : T}$$

$$\frac{\Gamma \vdash M : T}{\Gamma \vdash \mathsf{inl}\,M : T \oplus U} \qquad \frac{\Gamma \vdash M : U}{\Gamma \vdash \mathsf{inr}\,M : T \oplus U}$$

$$\frac{\Gamma \vdash M : T \oplus T' \quad \Gamma', x : T \vdash N : U \quad \Gamma', x : T' \vdash N' : U}{\Gamma, \Gamma' \vdash \mathsf{case}\,M\,\{\mathsf{inl}\,x \mapsto N; \mathsf{inr}\,x \mapsto N'\} : U}$$

$$\frac{\Gamma \vdash M : \mathbf{0}}{\Gamma, \Gamma' \vdash \mathsf{absurd}\,M : T} \qquad \frac{\Gamma \vdash M : T \quad \Gamma \vdash N : U}{\Gamma \vdash \langle M, N\rangle : T \,\&\, U}$$

$$\frac{\Gamma \vdash M : T \,\&\, U}{\Gamma \vdash \mathsf{fst}\,M : T} \qquad \frac{\Gamma \vdash M : T \,\&\, U}{\Gamma \vdash \mathsf{snd}\,M : U} \qquad \frac{}{\Gamma \vdash \langle\rangle : \top}$$

Values and Contexts

$$\begin{array}{rcl} V, W & ::= & x \mid \lambda^{\sigma} x.M \mid (V, W) \mid () \mid \mathsf{inl}\,V \mid \mathsf{inr}\,V \mid \langle V, W\rangle \mid \langle\rangle \\ \sigma & ::= & \{V_1/x_1, \ldots, V_n/x_n\} \\ & & \quad \text{where the } x_i \text{ are pairwise distinct} \\ E & ::= & [\,] \mid K\,E \mid E\,M \mid V\,E \\ & & \mid (E, M) \mid (V, E) \mid \mathsf{let}\ (x, y) = E\ \mathsf{in}\ M \\ & & \mid \mathsf{let}\ () = E\ \mathsf{in}\ M \\ & & \mid \mathsf{inl}\,E \mid \mathsf{inr}\,E \\ & & \mid \mathsf{case}\,E\,\{\mathsf{inl}\,x \mapsto N; \mathsf{inr}\,x \mapsto N'\} \\ & & \mid \mathsf{absurd}\,E \\ & & \mid \langle E, M\rangle \mid \langle V, E\rangle \mid \mathsf{fst}\,E \mid \mathsf{snd}\,E \end{array}$$

Reduction

$$\begin{array}{rcl} (\lambda^{\sigma} x.M)\,V & \longrightarrow_{\mathrm{v}} & M\{V/x \uplus \sigma\} \\ \mathsf{let}\ (x, y) = (V, W)\ \mathsf{in}\ M & \longrightarrow_{\mathrm{v}} & M\{V/x, W/y\} \\ \mathsf{let}\ () = ()\ \mathsf{in}\ M & \longrightarrow_{\mathrm{v}} & M \\ \mathsf{case}\,(\mathsf{inl}\,V)\,\left\{\begin{array}{l}\mathsf{inl}\,x \mapsto N;\\ \mathsf{inr}\,y \mapsto N'\end{array}\right\} & \longrightarrow_{\mathrm{v}} & N\{V/x\} \\ \mathsf{fst}\,\langle V, W\rangle & \longrightarrow_{\mathrm{v}} & V \\ \mathsf{snd}\,\langle V, W\rangle & \longrightarrow_{\mathrm{v}} & W \end{array}$$

$$E[M] \longrightarrow_{\mathrm{v}} E[M'] \qquad \text{if } M \longrightarrow_{\mathrm{v}} M'$$

Figure 1: Functional GV Syntax and Semantics.

the (entirely standard) definition of positivity and well-formedness conditions for types and type operators. As Functional GV is a linear calculus, the typing rule for $(\!|M|\!)$ mandates an empty environment: the evaluation of $(\!|M|\!)$ may require arbitrarily many copies of the expression M, and duplicating M would require duplicating its assumptions. Functional GV contains only linear assumptions; adding the exponential modality would introduce non-linear assumptions, which could be used in the bodies of catamorphisms.

The semantics of folds is given at the bottom of Figure 2. The extension of values and contexts is unsurprising. Each reduction of a fold $(\!|M|\!)$ amounts to one unrolling of the body M, and depends on the action of F on terms. With the exception of identity, each type constructor gives rise to both covariant (F) and contravariant functors (F^-). We have the expected typings that if $M : T \multimap U$ then $F(M) : F(T) \multimap F(U)$ and $F^-(M) : F(U) \multimap F(T)$. We use point-free notation for building functors over binary type constructors: for binary type constructor $*$, we write $F * G$ as shorthand for $X.F(X) * G(X)$.

Example. We turn to natural numbers as a characteristic example of recursive types. The definition of the type of naturals parallels the standard (intuitionistic) definition:

$$N(X) = \mathbf{1} \oplus X \qquad\qquad Nat = \mu N$$

and we can give familiar definitions of the constructors:

$$zero = \mathsf{in}\,(\mathsf{inl}\,()) \qquad\qquad succ = \lambda z.\mathsf{in}\,(\mathsf{inr}\,z)$$

Now consider a standard recursive definition of addition:

$$0 + y = y \qquad\qquad (S\,x) + y = S\,(x + y)$$

We can rewrite this definition as a curried function of one argument

$$plus\,0 = id \qquad\qquad plus\,(S\,x) = S \circ plus\,x$$

where id is the identity function and $\circ$ is function composition. Finally, this version can be expressed using a fold:

$$plus = (\!|\lambda x.\mathsf{case}\,x\,\{\mathsf{inl}\,() \mapsto id; \mathsf{inr}\,f \mapsto succ \circ f\}|\!)$$

The body of the fold has type $N(Nat \multimap Nat) \multimap (Nat \multimap Nat)$, so $plus$ has type $Nat \multimap Nat \multimap Nat$. The product of x and y, unlike their sum, cannot be computed without duplicating (in some way) either x or y. In an intuitionistic setting, we might accomplish this by capturing x in the body of the fold, and using it in each iteration. We cannot do the same in the linear setting. Instead, we will begin by demonstrating terms that duplicate and discard naturals. That is, we show that contraction and weakening are derivable for proposition Nat. This is an instance of a general result, due to Filinski [18], that contraction and weakening are derivable for the positive combinators in intuitionistic linear logic. In our setting this result is extended to include least fixed points μF.

$$\begin{array}{rl} dup = (\!|\lambda x.\mathsf{case}\,x\,\{ & \mathsf{inl}\,() \mapsto (zero, zero) \\ & \mathsf{inr}\,(y, z) \mapsto (succ\,y, succ\,z)\}|\!) \\ drop = (\!|\lambda x.\mathsf{case}\,x\,\{ & \mathsf{inl}\,() \mapsto () \\ & \mathsf{inr}\,() \mapsto ()\}|\!) \end{array}$$

We have that $dup : Nat \multimap Nat \times Nat$, where the output naturals are equal to the input natural, and $drop : Nat \multimap \mathbf{1}$, where we can then trivially eliminate the unit value. We can now implement multiplication:

$$\begin{array}{rl} body = \lambda x.\lambda y.\mathsf{case}\,y\,\{ & \mathsf{inl}\,() \mapsto (x, 0) \\ & \mathsf{inr}\,(x, y) \mapsto \mathsf{let}\,(x, x') = dup\,x\,\mathsf{in} \\ & \qquad (x', plus\,x\,y)\}; \\ times = \lambda x.\lambda y.\mathsf{let}\ (x', z) = (\!|body\,x|\!)\,y\ \mathsf{in}\ drop\,x'; z & \end{array}$$

The body of the fold has type $N(Nat \times Nat) \multimap Nat \times Nat$; in the inductive case, the input pair will be $(x, x(y-1))$, and the result is then (x, xy). We duplicate x at each step: one copy is added to the

Syntax

$$\begin{array}{lll} \text{Types} & & T, U ::= \cdots \mid X \mid \mu F \\ \text{Operators} & & F, G ::= X.T \\ \text{Terms} & & L, M, N ::= \cdots \mid \mathsf{in}\, M \mid (\!|M|\!) \end{array}$$

Typing

$$\frac{\Gamma \vdash M : F(\mu F)}{\Gamma \vdash \mathsf{in}\, M : \mu F} \qquad \frac{\vdash M : F(T) \multimap T}{\vdash (\!|M|\!) : \mu F \multimap T}$$

Values and Contexts

$$\begin{array}{rcl} V, W & ::= & \cdots \mid (\!|M|\!) \\ E & ::= & \cdots \mid \mathsf{in}\, E \end{array}$$

Covariant Functors

$$\begin{array}{rcl} (X.T)(M) & = & \lambda x.x \qquad X \text{ not free in } T \\ (X.X)(M) & = & M \\ (F \otimes G)(M) & = & \lambda x.\mathsf{let}\ (y,z) = x\ \mathsf{in}\ (F(M)\, y, G(M)\, z) \\ (F \multimap G)(M) & = & \lambda f.G(M) \circ f \circ F^-(M) \\ (F \oplus G)(M) & = & \lambda x.\mathsf{case}\, x\, \{\mathsf{inl}\, x \mapsto F(M)\, x; \mathsf{inr}\, x \mapsto G(M)\, x\} \\ (F \,\&\, G)(M) & = & \lambda x.\langle F(M)\, (\mathsf{fst}\, x), G(M)\, (\mathsf{snd}\, x)\rangle \\ (X.\mu F)(M) & = & (\!|\lambda x.\mathsf{in}\, ((X.F(\mu F))(M)\, x)|\!) \end{array}$$

Contravariant Functors

$$\begin{array}{rcl} (X.T)^-(M) & = & \lambda x.x \qquad X \text{ not free in } T \\ (F \times G)^-(M) & = & \lambda x.\mathsf{let}\ (y,z) = x\ \mathsf{in}\ (F^-(M)\, y, G^-(M)\, z) \\ (F \multimap G)^-(M) & = & \lambda f.G^-(M) \circ f \circ F(M) \\ (F \oplus G)^-(M) & = & \lambda x.\mathsf{case}\, x\, \{\mathsf{inl}\, x \mapsto F^-(M)\, x; \mathsf{inr}\, x \mapsto G^-(M)\, x\} \\ (F \,\&\, G)^-(M) & = & \lambda x.\langle F^-(M)\, (\mathsf{fst}\, x), G^-(M)\, (\mathsf{snd}\, x)\rangle \\ (X.\mu F)^-(M) & = & (\!|\lambda x.\mathsf{in}\, ((X.F(\mu F))^-(M)\, x)|\!) \end{array}$$

Reduction

$$(\!|M|\!)\ (\mathsf{in}\, V) \longrightarrow_{\mathrm{V}} M\ (F((\!|M|\!))\ V) \quad \text{if } M : F(A) \multimap A \text{ for some } A$$

Figure 2: Extending Functional GV with Recursion.

product, while the other copy appears in the result. The result of the $(\!|body\, x|\!)\, y$ is the pair (x, xy); we call *drop* to discard the last copy of x, and return xy.

2.3 Corecursion and Nontermination

This section describes a further extension of Functional GV with corecursive data types (νF), the greatest fixed point of the functor F. Our treatment is dual to that of recursive types: we introduce a type operator for νF for the greatest fixed point of positive functor F, and terms out, witnessing that it is an F-coalgebra, and the anamorphism $[\![M]\!]$, witnessing its finality. The full syntax and semantics of corecursive types are given in Figure 3. The typing rules and interpretation of corecursive types are dual to those for recursive types: where recursive types provide an iterated fold operation and a finite number of folding steps, corecursive types provide an iterated unfold operation and a finite number of unfolding steps. We restrict the type environment in unfolding to avoid duplicating linear resources. We extend the syntax of values with unfolds $[\![M]\!]$ both unapplied (of type $A \multimap \nu F$) and applied (of type νF). The treatment of greatest fixed points as functors is unsurprising; the reduction rule for anamorphisms unrolls the term M relying on the action of F as expected. Functional μGV is Functional GV extended with recursive and corecursive data types.

Example. As a canonical example of corecursive types, we consider streams of naturals.

$$S(X) = \mathbf{1} \,\&\, (Nat \otimes X) \qquad Stream = \nu S$$

Syntax

$$\begin{array}{lll} \text{Types} & & T ::= \cdots \mid \nu F \\ \text{Terms} & & M ::= \cdots \mid \mathsf{out}\, M \mid [\![M]\!] \end{array}$$

Typing

$$\frac{\Gamma \vdash M : \nu F}{\Gamma \vdash \mathsf{out}\, M : F(\nu F)} \qquad \frac{\vdash M : A \multimap F(A)}{\vdash [\![M]\!] : A \multimap \nu F}$$

Values and Contexts

$$\begin{array}{rcl} V, W & ::= & [\![M]\!] \mid [\![M]\!]\, V \mid \ldots \\ E & ::= & \mathsf{out}\, E \mid \ldots \end{array}$$

Covariant Functor

$$(X.\nu F)(M) = [\![\lambda x.(X.F(\nu F))(M)\ (\mathsf{out}\, x)]\!]$$

Contravariant Functor

$$(X.\nu F)^-(M) = [\![\lambda x.(X.F(\nu F))^-(M)\ (\mathsf{out}\, x)]\!]$$

Reduction

$$\mathsf{out}\ ([\![M]\!]\, V) \longrightarrow_{\mathrm{V}} F([\![M]\!])\ (M\, V)$$

Figure 3: Extending GV with Corecursion

The definition here differs from the typical intuitionistic definition: as our streams are linear, we need to include some provision for terminating the stream. Nevertheless, the choice of the length of the sum lies with the consumer of the stream, precisely dual to the case with recursive types, in which the choice lies with the producer. We can demonstrate this by showing a term of type νS, enumerating all the natural numbers:

$$upFrom = [\![\lambda x.\langle drop\, x, \mathsf{let}\ (y,z) = dup\, x\ \mathsf{in}\ (y, succ\, z)\rangle]\!]$$

Here the body of *nats* has type $Nat \multimap S(Nat)$, and so *upFrom* has type $Nat \multimap Stream$. A consumer of the stream can choose how many elements it reads; for example, here is a term that reads the first two naturals from a given stream:

$$\begin{array}{l} firstTwo\, s = \mathsf{let}\ (x,s) = \mathsf{snd}\ (\mathsf{out}\, s)\ \mathsf{in} \\ \qquad\qquad \mathsf{let}\ (y,s) = \mathsf{snd}\ (\mathsf{out}\, s)\ \mathsf{in} \\ \qquad\qquad \mathsf{fst}\ (\mathsf{out}\, s); (x,y) \end{array}$$

Nontermination. Freyd [19] observed that: a) the greatest and least fixed points of functors coincide in many denotational models of functional languages, and b) recognizing this coincidence gives an interpretation to many non-terminating recursive programs. One can apply this observation to Functional μGV by identifying the types μF and νF. Doing so has several consequences for the term language. Observe that out and in now compose (in either order), giving the identity. This gives rise to two new reduction rules to account for these compositions:

$$\mathsf{out}\ (\mathsf{in}\, V) \longrightarrow_{\mathrm{V}} V \qquad\qquad \mathsf{in}\ (\mathsf{out}\, V) \longrightarrow_{\mathrm{V}} V$$

It is now possible to compose folds and unfolds to define recursive computations. Such compositions are sometimes called hylomorphisms [29]. Hylomorphisms are accounted for by the following reduction:

$$(\!|N|\!)\ ([\![M]\!]\, V) \longrightarrow_{\mathrm{V}} N\ (F((\!|N|\!))\ (F([\![M]\!])\ (M\, V))).$$

Intuitively, each evaluation of a hylomorphism corresponds to one folding step and one unfolding step. Such an extension would break the logical interpretation of Functional GV, as all types are inhabited by the trivial hylomorphism. However, it does show a direct and intuitive connection between the (admittedly austere) setting of Functional GV and more practical programming languages.

3. Concurrent μGV

3.1 Communication and Concurrency

We now consider the concurrent fragment of μGV. The additional syntax of the concurrent fragment is listed at the top of Figure 4. Our primitive session types include input ($?T.S$), output ($!T.S$), and closed channels ($\mathsf{end}_?, \mathsf{end}_!$). Unlike many session type systems, but in keeping with logically-founded approaches, we have dual types for closed channels rather than a single, self-dual type end. (In our prior work [26] we discuss the semantic and logical consequences of providing a self-dual closed channel.) The remaining features of concurrent μGV can be encoded in terms of the primitive concurrent features, and the features of Functional μGV. These include selection ($S \oplus^! S'$), branching ($S \oplus^? S'$) and recursive session types ($\mu^! \mathcal{F}, \mu^? \mathcal{F}$). In traditional session typing notation, $\oplus^!$ is written as $\oplus$, $\oplus^?$ as &, $\mathsf{inl}^!$ as select inl, and $\mathsf{case}^?$ as offer. To avoid conflicting with the base features of Functional μGV and to emphasize the uniformity of our extensions, we adopt notation which makes explicit the direction of communication. For example, the ! denotes that $\mathsf{inl}^!$ sends a left injection along a channel. Note that fork is the only term that introduces new session-typed channels; the remaining session-typed constructs all consume channels. Thus, for example, the typing of $\mathsf{inl}^!$ is inverted from the typing of inl, eliminating the session type $S \oplus^! S'$.

Our treatment of recursive session types is also guided by initial algebra semantics. We introduce session type variables $\mathcal{X}$ and session functors $\mathcal{F}$. The argument and result of a session functor are both session types. We extend the standard notion of positivity to session functors and ordinary functors of session type (Figure 4). Just as we distinguished between consuming ($(\!|-|\!)$) and producing (in) values of recursive types, we distinguish between consuming and producing recursive communication. Thus, we have two dual constructors for recursive session types, $\mu^? F$ for consuming recursive communication and $\mu^! F$ for producing it. The terms inhabiting recursive session types are similar to those for recursive data types: $\mathsf{in}^! M$ unfolds one iteration of a recursive session type, while $(\!|M|\!)^?$ consumes a recursive session type; the production and consumption roles are indicated by direction of communication. The typing of $\mathsf{in}^! M$ reflects its role as an eliminator of session types, parallel to the typing of $\mathsf{inl}^!$ and $\mathsf{inr}^!$.

As for recursive session types, we have corecursive session types $\nu^! \mathcal{F}$ and $\nu^? \mathcal{F}$. The typing rule for $\mathsf{out}^? M$ is a direct reflection of the rule for $\mathsf{out}\, M$. As the communication primitives all consume terms of session type, $[\![-]\!]^!$ consumes a channel of type $\nu^! \mathcal{F}$, and returns the remaining (empty) expectations of the channel. The differences between recursive and corecursive session types are apparent both in the term formation rules and in the direction of communication: folds $(\!|M|\!)^?$ consume recursive communication, while unfolds $[\![M]\!]^!$ produce corecursive communication. Though their typing rules appear similar, the directions of communication in $\mathsf{in}^! M$ and $\mathsf{out}^? M$ are opposite.

The notion of duality is central to session types: if the process holding one end of a channel expects to send a value of some type along that channel, the process holding the other end should expect to receive a value of the same type. The dual $\overline{S}$ of session type S is defined in Figure 4. The dual of recursive ($\mu^! \mathcal{F}, \mu^? \mathcal{F}$) and corecursive ($\nu^! \mathcal{F}$, $\nu^? \mathcal{F}$) session types is defined in terms of the dualized session functor $\overline{\mathcal{F}}$. In the definition of $\overline{\mathcal{F}}$, note that we not only dualize the body of $\mathcal{F}$, but also the variable; this accounts for the duality between the two forms of recursion. We contrast this approach with standard approaches to duality for recursive session types in §3.2.

Promises. While aesthetically appealing, our formation of recursive session types seems somewhat awkward to use. The typing of $(\!|M|\!)^?$ requires that M transform $\mathcal{F}(S)$ (a session) into S (itself a session); that is, it flattens nested sessions into single sessions. In contrast, most uses of recursive session types transform the data values carried by the session into a result value, only incidentally relying on the nesting of sessions. We can relate these views by observing that μGV provides a natural notion of *promise*, and that promises allow us to treat arbitrary types as session types. Promises [28] introduce asynchrony between the computation of a value and its use; a promise of type T denotes a value of type T which may not yet have been computed. A promise of type T arises naturally in our setting as a channel of type $?T.\mathsf{end}_?$. We write $?T$ to abbreviate $?T.\mathsf{end}_?$, and define mappings between promises and values:

$$
\begin{array}{l}
un^? : ?T \multimap T \\
un^?\, M = \mathsf{let}\ (z, c) = \mathsf{receive}\, M\ \mathsf{in} \\
\qquad\qquad \mathsf{wait}\, c; z \\
en^? : T \multimap ?T \\
en^?\, M = \mathsf{fork}\ (\lambda x.\mathsf{send}\ (M, x))
\end{array}
$$

Syntax

$$
\begin{array}{lrl}
\text{Session types} & S ::= & !T.S \mid ?T.S \mid \mathsf{end}_! \mid \mathsf{end}_? \\
& \mid & S \oplus^! S' \mid \mathbf{0}^! \mid S \oplus^? S' \mid \mathbf{0}^? \\
& \mid & \mathcal{X} \mid \overline{\mathcal{X}} \mid \mu^! \mathcal{F} \mid \mu^? \mathcal{F} \mid \nu^! \mathcal{F} \mid \nu^? \mathcal{F} \\
\text{Types} & T, U ::= & \cdots \mid S \\
\text{Session functors} & \mathcal{F} ::= & \mathcal{X}.S \\
\text{Terms} & L, M, N ::= & \cdots \mid \mathsf{inl}^!\, M \mid \mathsf{inr}^!\, M \\
& \mid & \mathsf{case}^?\, L\, \{\mathsf{inl}\, x \mapsto M; \mathsf{inl}\, y \mapsto N\} \\
& \mid & (\!|M|\!)^? \mid \mathsf{in}^! M \mid \mathsf{out}^? M \mid [\![M]\!]^! \\
\text{Constants} & K ::= & \mathsf{send} \mid \mathsf{receive} \mid \mathsf{fork} \mid \mathsf{wait} \mid \mathsf{link}
\end{array}
$$

Duality

$$
\begin{array}{lll}
\overline{!T.S} = ?T.\overline{S} & \overline{\mathsf{end}_!} = \mathsf{end}_? & \overline{\mu^! \mathcal{F}} = \mu^? \overline{\mathcal{F}} \\
\overline{?T.S} = !T.\overline{S} & \overline{\mathsf{end}_?} = \mathsf{end}_! & \overline{\mu^? \mathcal{F}} = \mu^! \overline{\mathcal{F}} \\
\overline{S \oplus^! S'} = \overline{S} \oplus^? \overline{S'} & \overline{\mathbf{0}^!} = \mathbf{0}^? & \overline{\nu^! \mathcal{F}} = \nu^? \overline{\mathcal{F}} \\
\overline{S \oplus^? S'} = \overline{S} \oplus^! \overline{S'} & \overline{\mathbf{0}^?} = \mathbf{0}^! & \overline{\nu^? \mathcal{F}} = \nu^! \overline{\mathcal{F}} \\
\overline{\overline{S}} = S & \overline{X.S} = X.\overline{S} & \overline{\mathcal{X}.S} = \mathcal{X}.\overline{S}\{\overline{\mathcal{X}}/\mathcal{X}\}
\end{array}
$$

Positivity

$$
\begin{array}{l}
\xi \in \{X, \mathcal{X}\} \\
p(\xi.?T.S) = p(\xi.T) \wedge p(\xi.S), \quad \text{where } p \in \{neg, pos\} \\
pos(\xi.!T.S) = neg(\xi.T) \wedge pos(\xi.S) \\
neg(\xi.!T.S) = pos(\xi.T) \wedge neg(\xi.S)
\end{array}
$$

Typing

$$
\frac{\Gamma \vdash M : S \oplus^! S'}{\Gamma \vdash \mathsf{inl}^!\, M : S} \qquad
\frac{\Gamma \vdash M : S \oplus^! S'}{\Gamma \vdash \mathsf{inr}^!\, M : S'}
$$

$$
\begin{array}{r}
\mathsf{send} : T \otimes\, !T.S \multimap S \\
\mathsf{receive} : ?T.S \multimap T \otimes S \\
\mathsf{fork} : (S \multimap \mathsf{end}_!) \multimap \overline{S} \\
\mathsf{wait} : \mathsf{end}_? \multimap \mathbf{1} \\
\mathsf{link} : S \otimes \overline{S} \multimap \mathsf{end}_!
\end{array}
$$

$$
\frac{\Gamma \vdash L : S \oplus^? S' \quad \Gamma, x : S \vdash M : T \quad \Gamma, x : S' \vdash N : T}{\Gamma \vdash \mathsf{case}^?\, L\, \{\mathsf{inl}\, x \mapsto M; \mathsf{inr}\, x \mapsto N\} : T}
$$

$$
\frac{\Gamma \vdash L : \mathbf{0}^?}{\Gamma \vdash \mathsf{case}^?\, L\, \{\} : T} \qquad
\frac{\Gamma \vdash M : \mu^! \mathcal{F}}{\Gamma \vdash \mathsf{in}^! M : \mathcal{F}(\mu^! \mathcal{F})} \qquad
\frac{\vdash M : \mathcal{F}(S) \multimap S}{\vdash (\!|M|\!)^? : \mu^? \mathcal{F} \multimap S}
$$

$$
\frac{\Gamma \vdash M : \nu^? \mathcal{F}}{\Gamma \vdash \mathsf{out}^? M : \mathcal{F}(\nu^? \mathcal{F})} \qquad
\frac{\vdash L : S \multimap \mathcal{F}(\overline{S}) \multimap \mathsf{end}_!}{\vdash [\![L]\!]^! : S \multimap \nu^! \mathcal{F} \multimap \mathsf{end}_!}
$$

(Shaded terms and types, and their typing and duality, can be encoded in terms of the remaining terms and types.)

Figure 4: Concurrent μGV Terms and Typing

The operation $un^?M$ retrieves a value from promise M (blocking until it is available), while $en^?M$ constructs a new promise, already containing the value of M. Note that $un^? \circ en^?$ and $en^? \circ un^?$ are both observationally equivalent to the identity function (the second is a trivial example of channel forwarding). The dual of the type $?T$ is the type $!T.\mathsf{end}_!$, which we will abbreviate $!T$. We can also define an operation to introduce and eliminate channels of type $!T$ (that is, to provide results to unfulfilled promises):

$$\begin{array}{l} un^! : ((\overline{S} \multimap T) \otimes !T) \multimap S \\ un^!(L, M) = \mathsf{fork}\ (\lambda x.\mathsf{send}\ (L\,x, M)) \\ en^! : \overline{S} \multimap !S \\ en^!(M) = \mathsf{fork}\ (\lambda z.\mathsf{let}\ (c, z) = \mathsf{receive}\ z\ \mathsf{in} \\ \qquad\qquad\qquad\qquad \mathsf{wait}\ z; \mathsf{link}\ (M, c)) \end{array}$$

The appearance of the continuation type S may be surprising; this is a consequence of the different treatment of the continuation in send and receive. As μGV lacks polymorphism, we will write $un^?M$ to denote the substitution of M into the definition of $un^?$, rather than to denote an application in μGV, and similarly for the other definitions in this section.

Example: Recursive Sessions. We now present several examples of channels of naturals, building on our earlier representation of naturals numbers. We introduce type abbreviations for such channels:

$$NC(X) = \mathsf{end}_? \oplus^? ?Nat.X \qquad\qquad Nats = \mu^? NC$$

We begin with a term that sends two naturals along a given channel:

$$\begin{array}{l} twoNats : \overline{Nats} \multimap \mathsf{end}_! \\ twoNats = \lambda c.\mathsf{let}\ c = \mathsf{send}\ (zero, \mathsf{inr}^!\ (\mathsf{in}^!\ c))\ \mathsf{in} \\ \qquad\qquad\qquad \mathsf{let}\ c = \mathsf{send}\ (succ\ zero, \mathsf{inr}^!\ (\mathsf{in}^!\ c))\ \mathsf{in} \\ \qquad\qquad\qquad \mathsf{inl}^!\ (\mathsf{in}^!\ c) \end{array}$$

Now we present a slightly more interesting example. Given some starting natural n, we send the sequence $n, n-1, \ldots, 0$ along a channel. We rely on *Nat* itself being defined recursively.

$$\begin{array}{l} downFrom : \overline{Nats} \multimap \mathsf{end}_! \\ downFrom\ n = \mathsf{let}\ (n', k) = (\!|body|\!)\ n\ \mathsf{in}\ drop\ n'; k \\ \\ body : N(Nat \otimes (\overline{Nats} \multimap \mathsf{end}_!)) \multimap (Nat \otimes (\overline{Nats} \multimap \mathsf{end}_!)) \\ body\ z = \mathsf{case}\ z\ \{\mathsf{inl}\ () \quad \mapsto\ zc; \\ \qquad\qquad\qquad \mathsf{inr}\ (y, k)\ \mapsto\ sc\ y\ k\} \\ zc : Nat \otimes (\overline{Nats} \multimap \mathsf{end}_!) \\ zc = (zero, \lambda c.\mathsf{let}\ c = \mathsf{send}\ (zero, \mathsf{inr}^!\ (\mathsf{in}^! c))\ \mathsf{in} \\ \qquad\qquad \mathsf{inl}^!\ (\mathsf{in}^! c)) \\ sc : Nat \multimap (\overline{Nats} \multimap \mathsf{end}_!) \multimap (Nat \otimes (\overline{Nats} \multimap \mathsf{end}_!)) \\ sc\ y\ k = \mathsf{let}\ (y, y') = dup\ y\ \mathsf{in} \\ \qquad (succ\ y, \lambda c.k\ (\mathsf{send}\ (y', \mathsf{inr}^!\ (\mathsf{in}^! c)))) \end{array}$$

We have a similar challenge in defining *body* as we did in defining *times*: at each step of the recursion, we must both send a value along the channel and produce the same value for the next step. Observe that *body* has type $N(Nat \otimes (\overline{Nats} \multimap \mathsf{end}_!)) \multimap Nat \otimes (\overline{Nats} \multimap \mathsf{end}_!)$, computing both the next natural in the sequence and the function that sends it along a channel.

We can also write functions that consume channels of naturals. For a simple example, we could compute the sum of the naturals received along a channel.

$$\begin{array}{l} sum : Nats \multimap Nat \\ sum = (\!|\lambda c.\mathsf{case}^?\ c\ \{ \\ \qquad\qquad \mathsf{inl}\ c \mapsto \mathsf{wait}\ c; en^?\ zero \\ \qquad\qquad \mathsf{inr}\ c \mapsto \mathsf{let}\ (x, c) = \mathsf{receive}\ c\ \mathsf{in} \\ \qquad\qquad\qquad\qquad \mathsf{let}\ y = un^?\ c\ \mathsf{in} \\ \qquad\qquad\qquad\qquad en^?\ (plus\ x\ y)\}|\!)^? \end{array}$$

We wrap the running sum in a promise to lift it to session type; the body of the fold has type $NC(?Nat) \multimap ?Nat$, so *sum* has type $\mu^? NC \multimap ?Nat$. We could compose this with one of the producers above to compute a value, such as:

$$un^?\ (sum\ (\mathsf{fork}\ (downFrom\ \underline{4})))$$

where we write $\underline{n}$ to indicate the representation of natural n. This term will evaluate to $\underline{10}$.

We can also define channel transformers. For example, we could compute the running total of the stream, inserting the total (to that point) after each element.

$$\begin{array}{l} running : \overline{Nats} \multimap Nats \multimap \mathsf{end}_! \\ running\ c = un^?\ ((\!|\lambda c.\mathsf{case}^?\ c\ \{\mathsf{inl}\ c \mapsto done\ c; \\ \qquad\qquad\qquad\qquad\qquad \mathsf{inr}\ c \mapsto more\ c\}|\!)^?\ c)\ 0 \\ done\ c = en^?\ (\lambda z.\lambda d.drop\ z; \mathsf{link}\ (c, \mathsf{inl}^!\ (\mathsf{in}^! d))) \\ more\ c = \mathsf{let}\ (y, c) = \mathsf{receive}\ c\ \mathsf{in} \\ \qquad\qquad \mathsf{let}\ k = un^?\ c\ \mathsf{in} \\ \qquad\qquad \mathsf{let}\ (y, y') = dup\ y\ \mathsf{in} \\ \qquad\qquad en^?\ (\lambda z.\lambda d.\mathsf{let}\ (w, w') = dup\ (plus\ y'\ z)\ \mathsf{in} \\ \qquad\qquad\qquad\qquad \mathsf{let}\ d = \mathsf{send}\ (y, \mathsf{inr}^!\ (\mathsf{in}^! d))\ \mathsf{in} \\ \qquad\qquad\qquad\qquad \mathsf{let}\ d = \mathsf{send}\ (w, \mathsf{inr}^!\ (\mathsf{in}^! d))\ \mathsf{in} \\ \qquad\qquad\qquad\qquad k\ w'\ d) \end{array}$$

Note that the body of the fold has type $NC(?(Nat \multimap \overline{Nats} \multimap \mathsf{end}_!)) \multimap ?(Nat \multimap \overline{Nats} \multimap \mathsf{end}_!)$; the *Nat* argument stores the running sum, and so is initialized to zero by *running*.

Example: Corecursive Sessions. The treatment of corecursive session types is similar to that of recursive session types. Nevertheless, we give a short example of their use. The previous example showed a term *downFrom* that sent all the naturals below a starting value along a channel. With corecursive session types, we can give a term that (potentially) sends all the naturals above a starting value along a channel. We begin with the types; unlike the previous version, it is now the side sending naturals that offers to choice to continue:

$$NC'(X) = \mathsf{end}_! \oplus^? !Nat.X \qquad\qquad Nats' = \nu^! NC'$$

We can now define a process that sends an increasing stream of naturals:

$$\begin{array}{l} body = \lambda m.\lambda c.\mathsf{case}^?\ c\ \{\mathsf{inl}\ c \mapsto drop\ m; c; \\ \qquad\qquad\qquad\qquad\qquad \mathsf{inr}\ c \mapsto \mathsf{let}\ (m, m') = dup\ m\ \mathsf{in} \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{let}\ c = \mathsf{send}\ (m, c)\ \mathsf{in} \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{send}\ (succ\ m', c) \\ upFrom = [\![body]\!]^! \end{array}$$

As before, we rely on duplication and discard being defined for natural numbers. We have that $body : Nat \multimap NC'_!(Nat) \multimap \mathsf{end}_!$, and so $upFrom : Nat \multimap Nats' \multimap \mathsf{end}_!$. A simple consumer of such a channel might compute the sum of the first two values on the channel:

$$\begin{array}{l} firstTwo\ c = \mathsf{let}\ (x, c) = \mathsf{receive}\ (\mathsf{inr}^!\ (\mathsf{out}^? c))\ \mathsf{in} \\ \qquad\qquad\qquad \mathsf{let}\ (y, c) = \mathsf{receive}\ (\mathsf{inr}^!\ (\mathsf{out}^? c))\ \mathsf{in} \\ \qquad\qquad\qquad \mathsf{wait}\ (\mathsf{inl}^!\ (\mathsf{out}^? c)); x + y \end{array}$$

3.2 Recursion, Duality, and Session Types

We relate our statement of duality to previous accounts of recursive session types. Most existing approaches use equirecursive, self-dual recursive session types, and do not include dualized type variables. Our system lacks self-dual constructs, and so cannot encode self-dual recursive types. However, we can imagine extending our language with a construct $\mu^{\mathsf{S}}\mathcal{F}$ such that $\overline{\mu^{\mathsf{S}}\mathcal{F}} = \mu^{\mathsf{S}}\overline{\mathcal{F}}$.

Honda et al. [23] originally proposed recursive session types for a system of *first order* session types in which messages did not

include channel names. Duality was given by $\overline{\mu^{\mathsf{S}} X.T} = \mu^{\mathsf{S}} X.\overline{T}$, where $\overline{X} = X$. To distinguish this notion from ours, we write $\mathit{naive}(T)$ instead of $\overline{T}$. In contrast, our approach gives $\overline{\mu^{?} X.T} = \mu^{?} X.\overline{T}\{\overline{X}/X\}$. It is not hard to see that logical duality coincides with naive duality for first-order session types. Intuitively, if $\mu^{\mathsf{S}} X.T$ is first-order, then if we compute $\mu^{\mathsf{S}} X.\overline{T}\{\overline{X}/X\}$ each instance of X in T will first be dualised by $\overline{T}$ and then again by the substitution $\{\overline{X}/X\}$.

Independently, Bono and Padovani [10] and Bernardi and Hennessy [6] observe that naive duality is not enough for *higher-order* session types, that is, session types with support for delegation. Consider $S = \mu^{\mathsf{S}} X.?X.X$. The logical dual of S is $\mu^{\mathsf{S}} X.!(\mu^{\mathsf{S}} X.?X.X).X$, whereas the naive dual of S is $\mu^{\mathsf{S}} X.!X.X$, which is (equirecursively) equivalent to $\mu^{\mathsf{S}} X.!(\mu^{\mathsf{S}} X.!X.X).X$. It is not difficult to show that the logical dual yields the correct behaviour, whereas the naive dual does not. They (each) proposed a new definition of duality for recursive session types, using a selective form of substitution which applies only inside carried types. Later, Bernardi and Hennessy [7] observed that even this approach fails on examples such as $\mu^{\mathsf{S}} X.\mu^{\mathsf{S}} Y.?Y.X$. They propose converting each recursive session type into a so-called *m-closed* recursive session type before applying native duality. A recursive session type $\mu^{\mathsf{S}} X.T$ is m-closed if X does not occur free inside a carried type in T. It is straightforward to show that every recursive session type is (equirecursively) equivalent to an m-closed one, and that, as they they are essentially first-order, naive duality and logical duality coincide on m-closed recursive session types.

Duality for recursive session types clearly needs to be treated carefully. We are encouraged that our definition coincides with the state of the art for equirecursive self-dual session types. We believe that this also shows the value of our deconstruction of recursive session types into well-understood primitives: we are guided immediately to a correct, compact, and general definition of duality.

Remark. Dualized session type variables are redundant in equirecursive session types, as every session type $\mu^{\mathsf{S}} X.T$ is equivalent to $\mu^{\mathsf{S}} X.T\{\mu^{\mathsf{S}} Y.T\{X/\overline{Y}\}/\overline{X}\}$, where Y is a fresh type variable. However, dualized session type variables do yield a cleaner compositional definition of duality.

4. Communication with Concurrency

4.1 Encoding Concurrent Features

Our prior work on GV [26] focuses on keeping the core language as simple as possible. For example, rather than include branching and choice in the concurrent semantics directly, a choice can be encoded as the promise of a (data type) sum. Kobayashi et al [24] and Dardha et al. [17] make similar use of linear promises to relate data types and session types in π-calculi. We extend this view to include recursive session types. There are two challenges in doing so: a) we must encode session functors and their use of session type variables, and b) we must encode their fixed points.

Our translation is given by the homomorphic extension of the rules in Figure 5. We underline those portions of the translation that introduce purely administrative reduction. The session functor $\mathcal{F}$ is translated to the ordinary function $\mathcal{F}_{?}$, using promises to lift an ordinary type variable to session type. This approach naturally accounts for the use of dualized session type variables; for example, if $\mathcal{F}(\mathcal{X}) = !\overline{\mathcal{X}}.\mathcal{X}$, then we have that $\mathcal{F}_{?}(X) = !(\overline{?X}).?X = !(!X).?X$. Recursive and corecursive session types are interpreted as promises of recursive and corecursive data types, and the interpretation of their terms is directed by the interpretation of their types. The definitions of $(\!|M|\!)^{?}$ and $[\![M]\!]^{!}$ may seem surprisingly complicated. In fact, we can present a different form of session-typed catamorphisms and anamorphisms, directly encoded in terms of recursive types:

$$(\!|M|\!)^{\mathsf{S}}\, N = (\!|M|\!)\ (un^{?} M)$$
$$[\![L]\!]^{\mathsf{S}}\, M\, N = \mathsf{send}\ ([\![\lambda x.\mathsf{fork}\ (\lambda c.L\, x\, c)]\!] M, N)$$

with the following typing rule, which exchanges the restriction to session functors for a direct use of their encoding:

$$\frac{\vdash M : \mathcal{F}_{?}(T) \multimap T}{\vdash (\!|M|\!)^{\mathsf{S}} : \mu^{?}\mathcal{F} \multimap T} \qquad \frac{\vdash M : S \multimap \mathcal{F}_{!}(\overline{S}) \multimap \mathsf{end}_{!}}{\vdash [\![M]\!]^{\mathsf{S}} : S \multimap \nu^{?}\mathcal{F} \multimap \mathsf{end}_{!}}$$

We can see that the encoding of $(\!|-|\!)^{?}$ is an instance of $(\!|-|\!)^{\mathsf{S}}$, that the encoding of $[\![-]\!]^{!}$ is an instance of $[\![-]\!]^{\mathsf{S}}$, and that our examples can be written directly using the alternative forms (removing calls to $en^{?}$ as necessary). Nevertheless, we have preferred $(\!|-|\!)^{?}$ and $[\![-]\!]^{!}$ as they do not rely on details of our encoding and are closer to the algebraic intuition.

Session functors

$$\mathcal{F}_{?} = X.\mathcal{Q}[\![\mathcal{F}(?X)]\!] \qquad \mathcal{F}_{!} = X.\mathcal{Q}[\![\mathcal{F}(!X)]\!]$$

Session types

$$\begin{array}{ll} \mathcal{Q}[\![S \oplus^{!} S']\!] = !(\overline{\mathcal{Q}[\![S]\!]} \oplus \overline{\mathcal{Q}[\![S']\!]}) & \mathcal{Q}[\![\mu^{?}\mathcal{F}]\!] = ?\mu\mathcal{F}_{?} \\ \mathcal{Q}[\![S \oplus^{?} S']\!] = ?(\mathcal{Q}[\![S]\!] \oplus \mathcal{Q}[\![S']\!]) & \mathcal{Q}[\![\mu^{!}\mathcal{F}]\!] = !\mu\overline{\mathcal{F}}_{?} \\ \mathcal{Q}[\![\mathbf{0}^{?}]\!] = ?0 & \mathcal{Q}[\![\nu^{!}\mathcal{F}]\!] = !\nu\mathcal{F}_{!} \\ \mathcal{Q}[\![\mathbf{0}^{!}]\!] = !0 & \mathcal{Q}[\![\nu^{?}\mathcal{F}]\!] = ?\nu\overline{\mathcal{F}}_{!} \end{array}$$

Terms

$$\begin{array}{rl} \mathcal{Q}[\![\ell^{!}\, M]\!] & = \underline{un^{!}}(\lambda x.\ell\, x, \mathcal{Q}[\![M]\!]) \\ \mathcal{Q}\left[\!\!\left[\mathsf{case}^{?}\ L\ \left\{\begin{array}{l}\mathsf{inl}\, x \mapsto M; \\ \mathsf{inr}\, x \mapsto N\end{array}\right\}\right]\!\!\right] & = \mathsf{case}(\underline{un^{?}}\,\mathcal{Q}[\![L]\!])\ \left\{\begin{array}{l}\mathsf{inl}\, x \mapsto \mathcal{Q}[\![M]\!] \\ \mathsf{inr}\, x \mapsto \mathcal{Q}[\![N]\!]\end{array}\right\} \\ \mathcal{Q}[\![\mathsf{in}^{!}\, M]\!] & = \underline{un^{!}}\ (\lambda x.\mathsf{in}\, x, \mathcal{Q}[\![M]\!]) \\ \mathcal{Q}[\![(\!|M|\!)^{?}\, N]\!] & = (\!|\lambda y.\underline{en^{?}}\ (\mathcal{Q}[\![M]\!]\ (\underline{\mathcal{Q}[\![\mathcal{F}]\!](un^{?})}\, y))|\!) \\ & \quad\ (\underline{un^{?}}\ \mathcal{Q}[\![N]\!]) \\ \mathcal{Q}[\![\mathsf{out}^{?} M]\!] & = \mathsf{out}\ (\underline{un^{?}}\ \mathcal{Q}[\![M]\!]) \\ \mathcal{Q}[\![[\![L]\!]^{!}\, M\, N]\!] & = [\![\lambda y.\underline{\mathcal{Q}[\![\mathcal{F}]\!](en^{!})} \\ & \quad\ (\underline{un^{!}}\ (\lambda c.\mathsf{fork}\ (\lambda d.\mathcal{Q}[\![L]\!]\, d\, c), \\ & \quad\ y))]\!] \\ & \quad\ \mathcal{Q}[\![M]\!]\ (\underline{en^{!}}\ \mathcal{Q}[\![N]\!]) \end{array}$$

Figure 5: Translation of μGV concurrency features into core μGV

4.2 Concurrent Semantics

We give a concurrent semantics of μGV, building on the small-step operational semantics for Functional μGV given in the last section. As recursive session types and their terms can be encoded in terms of the core concurrency features, our semantics is mostly unchanged from that of our prior work [26]. Figures 6 and 7 give the syntax and typing of configurations and configuration contexts; we will write $\Gamma \vdash C : T$ to denote that there is some ϕ such that $\Gamma \vdash^{\phi} C : T$. Figure 8 gives reductions and configuration equivalence. Because of the importance of promises in our interpretation of μGV's concurrent features, we give special cases of the functor map for the promise functor. These have the same behavior as that given in the general case, but expose potentially administrative reductions sooner. Our treatment of link repairs a defect in our prior account of GV and restores the diamond property for GV's concurrent semantics.

The concurrent semantics of μGV is defined by a reduction relation on collections of parallel threads, called configurations. The syntax of μGV configurations is given in Figure 6, and includes threads, name restriction and composition of configurations. Be-

Configurations	$C ::= \phi M \mid (\mathsf{new}\, x)C$
	$\mid z = x \leftrightarrow y \mid C \parallel C'$
Flags	$\phi ::= \circ \mid \bullet$
Configuration contexts	$D ::= [\,] \mid (\mathsf{new}\, x)D \mid C \parallel D$
Thread evaluation contexts	$H ::= \phi E$

Figure 6: Configurations and Contexts.

$$\frac{\Gamma \vdash M : T}{\Gamma \vdash^{\bullet} \bullet M : T} \qquad \frac{\Gamma \vdash M : \mathsf{end}_!}{\Gamma \vdash^{\circ} \circ M : \mathsf{end}_!} \qquad \frac{\Gamma, x : S^\sharp \vdash^{\phi} C : T}{\Gamma \vdash^{\phi} (\mathsf{new}\, x)C : T}$$

$$\frac{}{x : S, y : \overline{S}, z : \mathsf{end}_? \vdash^{\circ} z = x \leftrightarrow y : \mathsf{end}_!}$$

$$\frac{\Gamma, x : S \vdash^{\phi} C : T \quad \Gamma', x : \overline{S} \vdash^{\circ} C' : \mathsf{end}_!}{\Gamma, \Gamma', x : S^\sharp \vdash^{\phi} C \parallel C' : T}$$

Figure 7: Configuration Typing.

cause μGV is a functional language, we distinguish the "main" thread $\bullet M$, which we expect to compute the result of the computation, from the child threads $\circ M$. We give a typing judgment for configurations, based on the type system for the linear π-calculus [24] but with two significant differences. First, we ensure that there is at most one main thread. This constraint is enforced by the flags ($\circ$ and $\bullet$) on the derivations: a derivation $\Gamma \vdash^{\bullet} C : T$ indicates that configuration C contains the main thread (returning a value of type T), while $\Gamma \vdash^{\circ} C : \mathsf{end}_!$ indicates that C does not. Second, we require that exactly one channel is shared at each composition of processes. This latter constraint is sufficient to guarantee deadlock freedom and progress, as we show in §5.

Theorem 1 (Diamond property)**.** *If $\Gamma \vdash C : T$, $C \equiv \longrightarrow \equiv C_1$, and $C \equiv \longrightarrow \equiv C_2$, then either $C_1 \equiv C_2$ or there exists C_3 such that $C_1 \equiv \longrightarrow \equiv C_3$, and $C_2 \equiv \longrightarrow \equiv C_3$.*

This proof extends to any deterministic extension of the core functional calculus, such as the addition of the exponential modality or polymorphism. The reader may be concerned that the WAIT rule does not apply in the case that x is returned from the main thread, and similarly for z in the LINK1 rule. However, these cases can never occur in a closed, well-typed configuration.

The other metatheoretic properties established by in our prior work hold here as well. In particular, reduction in μGV preserves typing.

Theorem 2. *If $\Gamma \vdash^{\phi} C : T$ and $C \longrightarrow C'$ then $\Gamma \vdash^{\phi} C' : T$.*

While typing is not preserved by configuration equivalence, reduction never produces or relies on ill-typed states.

Theorem 3. *If $\Gamma \vdash C_1 : T$, $C_1 \equiv C_2$, and $C_2 \longrightarrow C_2'$, then there is some C_1' such that $C_2' \equiv C_1'$, $C_1 \longrightarrow C_1'$ and $\Gamma \vdash C_1' : T$.*

We have encoded recursive session types using features of Functional μGV, and so they do not appear in the concurrent semantics directly. We would like to confirm that their encoding matches the intuition of the original, unencoded forms. That is, we hope that a configuration $H[(\!|M|\!)^?\, x] \parallel H'[\mathsf{in}^!\, x]$ reduces to $H[M\,(\mathcal{F}((\!|M|\!)^?)\, x)] \parallel H'[x]$. This reduction is blocked by the administrative steps introduced in the encoding of $(\!|-|\!)^?$. However, we can show that it holds if we can suitably ignore administrative reductions. To do so, we adapt a notion of weak bisimulation to our setting. Unlike standard presentations of concurrency, all μGV reductions are internal. Therefore, ignoring all internal reductions would trivially identify all processes that compute the same results. We intend a finer characterization, in which we ignore only administrative reductions. We have already identified (by underlining) the relevant sources of administrative reductions. We say that a reduction is administrative ($\underline{\longrightarrow}$) if all the reduced subexpressions are underlined. For example, the reduction of $H[\underline{\mathsf{send}\,(V,x)}] \parallel H'[\underline{\mathsf{receive}\, x}]$ to $H[x] \parallel H'[(V,x)]$ is administrative, but the reduction of $H[\underline{\mathsf{send}\,(M,x)}]$ to $H[\underline{\mathsf{send}\,(M',x)}]$ is not (unless M is itself identified as administrative). We write $\underline{\longrightarrow}^\star$ for the reflexive, transitive closure of $\underline{\longrightarrow}$, and write $C \Longrightarrow C'$ to denote $C \underline{\longrightarrow}^\star \longrightarrow \underline{\longrightarrow}^\star C'$. Finally, we can adapt the standard notion of weak bisimulation to our setting.

Definition 4. A relation $\mathcal{R}$ on configurations is an administrative weak bisimulation if, for each $C_1 \mathcal{R} C_2$, whenever $C_1 \Longrightarrow C_1'$, then there is a C_2' such that $C_2 \Longrightarrow C_2'$ and $C_1' \mathcal{R} C_2'$, and similarly for reduction from C_2. We define administrative weak bisimiliarity $\underline{\approx}$ to be the union of all administrative weak bisimulations.

We can now relate the encoding of recursive session types to their expected semantics:

Theorem 5.

1. *If $\vdash M : \mathcal{F}(S) \multimap S$, then*
$$(\mathsf{new}\, x)(H[(\!|M|\!)^?\, x] \parallel H'[\mathsf{in}^!\, x]) \longrightarrow^+ \underline{\approx} (\mathsf{new}\, x)(H[M\,(\mathcal{F}((\!|M|\!)^?)\, x)] \parallel H'[x])$$
2. *If $\vdash M : S \multimap \mathcal{F}(S)$, then*
$$(\mathsf{new}\, x)(H[[\![M]\!]^!\, V\, x] \parallel H'[\mathsf{out}^?\, x]) \longrightarrow^+ \underline{\approx} (\mathsf{new}\, x)(H[\mathcal{F}([\![M]\!]^!)\,(M\, V))\, x] \parallel H'[x])$$

The key observations to establishing this result are that $un^?$ and $en^?$ introduce only incidental additional concurrency.

Lemma 6.

1. $(\mathsf{new}\, x)(H[\underline{un^?\, x}] \parallel H'[\underline{un^!}\,(\lambda x.M, x)]) \underline{\approx} (\mathsf{new}\, x)(H[M] \parallel H'[x])$
2. $E[\underline{\mathcal{F}(un^?)}\,(\mathcal{F}(\lambda x.\underline{en^?}\,(M\, x))\, N)] \underline{\approx} E[\mathcal{F}(M)\, N]$
3. $E[\mathcal{F}(\lambda x.\underline{un^!}\,(\lambda x.M, x))\,(\underline{\mathcal{F}(en^!)}\, N)] \underline{\approx} E[\mathcal{F}(M)\, N]$

The first is entirely straightforward, the second and third can be shown by induction on the structure of $\mathcal{F}$. The theorem follows directly from the lemmas and the definition of reduction.

5. Communication without Concurrency

We now show, via a CPS translation, that reduction in μGV can be simulated by reduction in Functional μGV. We begin with a standard left-to-right call-by-value CPS translation from the core calculus into itself (Figures 9 and 10), where R is a fixed return type. In the rest of this subsection, we extend the CPS translation to session types and show that the CPS translation preserves reduction. As a corollary, we obtain that μGV is strongly normalising.

Following Danvy and Nielson [15], we can mechanically transform the naive CPS translation $\mathcal{N}[\![-]\!]$ of Figure 10 into a compositional first-order one-pass CPS transformation $\mathcal{K}[\![-]\!]$. By carefully distinguishing between values and non-values, the one-pass translation ensures that (most) administrative redexes are contracted by the translation itself. Contracting these redexes enables a tight simulation result (Theorem 10). Due to lack of space, we omit the (entirely standard) details of the one-pass variant of the translation.

Figure 11 gives the CPS translation of concurrent μGV. The translations of send, fork, and link depend on the polarities (input or output) of their arguments and results. We use subscripts to distinguish output and input session types. To give a compositional

Covariant Functors

$$
\begin{aligned}
(!F.G)(M) &= \lambda c.\mathsf{fork}\,(\lambda d.\mathsf{let}\,(z,d) = \mathsf{receive}\,d\,\mathsf{in}\\
&\qquad\qquad \mathsf{let}\,c = \mathsf{send}\,(F^{-}(M)\,z, c)\,\mathsf{in}\\
&\qquad\qquad \mathsf{link}\,(\overline{G}(M)\,d, G(M)\,c))\\
(?F.G)(M) &= \lambda c.\mathsf{fork}\,(\lambda d.\mathsf{let}\,(z,c) = \mathsf{receive}\,c\,\mathsf{in}\\
&\qquad\qquad \mathsf{let}\,d = \mathsf{send}\,(F(M)\,z, d)\,\mathsf{in}\\
&\qquad\qquad \mathsf{link}\,(\overline{G}(M)\,d, G(M)\,c))\\
(?F.\mathsf{end}_?)(M) &= \lambda c.\underline{en^?}\,(F(M)\,(\underline{un^?}\,c))
\end{aligned}
$$

Contravariant Functors

$$
\begin{aligned}
(!F.G)^{-}(M) &= \lambda c.\mathsf{fork}\,(\lambda d.\mathsf{let}\,(z,d) = \mathsf{receive}\,d\,\mathsf{in}\\
&\qquad\qquad \mathsf{let}\,c = \mathsf{send}\,(F(M)\,z, c)\,\mathsf{in}\\
&\qquad\qquad \mathsf{link}\,(\overline{G}^{-}(M)\,d, G^{-}(M)\,c))\\
(?F.G)^{-}(M) &= \lambda c.\mathsf{fork}\,(\lambda d.\mathsf{let}\,(z,c) = \mathsf{receive}\,c\,\mathsf{in}\\
&\qquad\qquad \mathsf{let}\,d = \mathsf{send}\,(F^{-}(M)\,z, d)\,\mathsf{in}\\
&\qquad\qquad \mathsf{link}\,(\overline{G}^{-}(M)\,d, G^{-}(M)\,c))\\
(?F.\mathsf{end}_?)^{-}(M) &= \lambda c.\underline{en^?}\,(F^{-}(M)\,(\underline{un^?}\,c))
\end{aligned}
$$

Configuration Equivalence

$$
\begin{array}{lll}
H[\mathsf{link}\,(x,y)] \equiv H[\mathsf{link}\,(y,x)] & C \parallel C' \equiv C' \parallel C & C \parallel (\mathsf{new}\,x)C' \equiv (\mathsf{new}\,x)(C \parallel C'), \quad \text{if } x \notin fv(C)\\
z = x \leftrightarrow y \equiv z = y \leftrightarrow x & C_1 \parallel (C_2 \parallel C_3) \equiv (C_1 \parallel C_2) \parallel C_3 & D[C] \equiv D[C'], \quad \text{if } C \equiv C'
\end{array}
$$

Configuration Reduction

$$
\begin{array}{lll}
\textsc{Send} & (\mathsf{new}\,x)(H[\mathsf{receive}\,x] \parallel H'[\mathsf{send}\,(V,x)]) \longrightarrow (\mathsf{new}\,x)(H[(V,x)] \parallel H'[x]) & \\
\textsc{Fork} & H[\mathsf{fork}\,(\lambda^{\sigma}y.M)] \longrightarrow (\mathsf{new}\,x)(H[x] \parallel \circ M(\{x/y\} \uplus \sigma)), & x \text{ fresh}\\
\textsc{Wait} & (\mathsf{new}\,x)(H[\mathsf{wait}\,x] \parallel \circ x) \longrightarrow H[()] & \\
\textsc{Link0} & H[\mathsf{link}\,(x,y)] \longrightarrow (\mathsf{new}\,z)(z = x \leftrightarrow y \parallel H[z]), & z \text{ fresh}\\
\textsc{Link1} & (\mathsf{new}\,z\,x)((z = x \leftrightarrow y \parallel \circ z) \parallel \phi M) \longrightarrow \phi M\{y/x\} &
\end{array}
$$

$$
\textsc{LiftV}\ \frac{M \longrightarrow_{\mathrm{V}} M'}{\phi M \longrightarrow \phi M'} \qquad \textsc{Lift}\ \frac{C \longrightarrow C'}{D[C] \longrightarrow D[C']}
$$

Figure 8: Concurrent Semantics of μGV: Functors, Equivalences, and Reductions.

translation of configurations, we restrict attention to a canonical class of configurations. We write $C_1 \parallel_x C_2$ to denote a parallel composition in which channel x has input session type in C_1 and the dual output session type in C_2. We say that a configuration C is *well-oriented* ($WO(C)$) if all of the parallel compositions in C are of this form, and in any link configuration $z = x \leftrightarrow y$ in C, x has input session type. Without loss of generality, we need only consider reduction on well-oriented configurations.

Lemma 7. *If $\Gamma \vdash C : T$, then there exists well-oriented $C' \equiv C$.*

The translation of the main thread is the only place the continuation is actually used. The translation of a child thread supplies the identity continuation, which is well-typed as child threads always have type $\mathsf{end}_!$. Name restrictions themselves are ignored, but names are used in the translation of well-oriented parallel composition. It is straightforward to verify that the CPS translation preserves typing.

Theorem 8 (Type soundness)**.**

1. *If $\Gamma \vdash M : T$, then $\mathcal{K}[\![\Gamma]\!] \vdash \mathcal{K}[\![M]\!] : (\mathcal{K}[\![T]\!] \multimap R) \multimap R$.*
2. *If $\Gamma \vdash C : T$ and C is well-oriented, then $\mathcal{K}[\![\Gamma]\!] \vdash \mathcal{K}[\![C]\!] : (\mathcal{K}[\![T]\!] \multimap R) \multimap R$.*

To reason by induction over the reduction rules, which are defined in terms of evaluation contexts and configuration contexts, we extend the CPS translation to contexts (Figure 12). Evaluation contexts are interpreted as functions. The CPS translation of a configuration context takes two arguments. The first argument is a meta-level function, which we instantiate with the CPS translation of an appropriate configuration. The second is a continuation. The CPS translation respects decomposition of contexts.

Lemma 9.

1. *If $E \neq [\,]$ and $\Gamma \vdash E[I] : T$, then $\mathcal{K}[\![E[I]]\!]k = \mathcal{I}[\![I]\!](\lambda x.\mathcal{E}[\![E]\!]k)$.*
2. *If $\Gamma \vdash D[C]$ and $D[C]$ is well-oriented, then $\mathcal{K}[\![D[C]]\!]k = \mathcal{D}[\![D]\!]\,\mathcal{K}[\![C]\!]\,k$.*

To simulate all reduction paths in μGV by reduction in Functional μGV, we must allow reduction under lambda abstractions. Otherwise, we would be limited to a single schedule in which order of communication is determined by the outermost input communication. For an intuition of this schedule, consider the translation of $C \parallel_x C'$. Reduction in C' cannot proceed until C is ready to receive a result along x, even if C' could perform some internal communication. Allowing reduction under lambda abstractions allows all valid schedules to be simulated. We define $M \rightsquigarrow N$ by:

$$
\frac{M \longrightarrow_{\mathrm{V}} N}{M \rightsquigarrow N} \qquad \frac{M \rightsquigarrow N}{\lambda^{\sigma}x.M \rightsquigarrow \lambda^{\sigma}x.N}
$$

The following theorem states that the CPS translation simulates μGV reduction.

Theorem 10 (Simulation)**.**

1. *If $\Gamma \vdash M : T$ and $M \longrightarrow N$, then $\mathcal{K}[\![M]\!]k \rightsquigarrow^{+} \mathcal{K}[\![N]\!]k$.*
2. *If $\Gamma \vdash C : T$ and $C \longrightarrow C'$, then there exist well-oriented C'', C''' with $C'' \equiv C$ and $C''' \equiv C'$ such that $\mathcal{K}[\![C'']\!]k \rightsquigarrow^{+} \mathcal{K}[\![C''']\!]k$.*

Thus we can simulate concurrent communication using only Functional μGV. As a corollary, we obtain that μGV is strongly normalising.

Theorem 11 (Strong normalization)**.** *If $\Gamma \vdash C : T$, then there are no infinite $\equiv\!\!\longrightarrow\!\!\equiv$ sequences starting from C.*

The proof follows immediately from Theorem 10 and the quite standard result that Functional μGV (linear λ-calculus with positive (co)recursive data types) is strongly normalising. (To show the latter, map functional μGV into System F, forgetting linearity, and encoding the positive (co)recursive data types using polymorphism.) An immediate consequence is that our calculus is free from livelock; that is, that there are no oscillating sequences of configurations that diverge.

The strong normalization result straightforwardly extends to the extension of μGV with the exponential modality [26], and to the setting where we allow reduction under lambdas in the source calculus.

$$\mathcal{K}[\![T \multimap U]\!] = \mathcal{K}[\![T]\!] \multimap (\mathcal{K}[\![U]\!] \multimap R) \multimap R$$

$$\begin{array}{ll} \mathcal{K}[\![T \otimes U]\!] = \mathcal{K}[\![T]\!] \otimes \mathcal{K}[\![U]\!] & \mathcal{K}[\![\mathbf{1}]\!] = \mathbf{1} \\ \mathcal{K}[\![T \oplus U]\!] = \mathcal{K}[\![T]\!] \oplus \mathcal{K}[\![U]\!] & \mathcal{K}[\![\mathbf{0}]\!] = \mathbf{0} \\ \mathcal{K}[\![T \,\&\, U]\!] = \mathcal{K}[\![T]\!] \,\&\, \mathcal{K}[\![U]\!] & \mathcal{K}[\![\top]\!] = \top \end{array}$$

$$\begin{array}{ll} \mathcal{K}[\![\mu X.T]\!] = \mu X.\mathcal{K}[\![T]\!] & \\ \mathcal{K}[\![\nu X.T]\!] = \nu X.\mathcal{K}[\![T]\!] & \mathcal{K}[\![X]\!] = X \end{array}$$

Figure 9: CPS Translation for Core Types

$$\begin{array}{c} \mathcal{N}[\![x]\!]k = k\ x \\ \mathcal{N}[\![K\ M]\!]k = \mathcal{N}[\![M]\!](\lambda x.\mathcal{N}[\![K]\!]\ x\ k) \\ \mathcal{N}[\![\lambda^{\{\vec{V}/\vec{z}\}}x.M]\!]k = k\ (\lambda x\ k.\mathcal{N}[\![\vec{V}]\!](\lambda\vec{z}.\mathcal{N}[\![M]\!]k)) \\ \mathcal{N}[\![M\ N]\!]k = \mathcal{N}[\![M]\!](\lambda x.\mathcal{N}[\![N]\!](\lambda y.x\ y\ k)) \\ \mathcal{N}[\![(M,N)]\!]k = \mathcal{N}[\![M]\!](\lambda x.\mathcal{N}[\![N]\!](\lambda y.k\ (x,y))) \\ \mathcal{N}[\![\mathsf{let}\ (x,y) = M\ \mathsf{in}\ N]\!]k = \mathcal{N}[\![M]\!](\lambda z.\mathsf{let}\ (x,y) = z\ \mathsf{in}\ \mathcal{N}[\![N]\!]k) \\ \mathcal{N}[\![()]\!]k = k\ () \\ \mathcal{N}[\![\mathsf{let}\ () = M\ \mathsf{in}\ N]\!]k = \mathcal{N}[\![M]\!](\lambda z.\mathsf{let}\ () = z\ \mathsf{in}\ \mathcal{N}[\![N]\!]k) \\ \mathcal{N}[\![\mathsf{inl}\,M]\!]k = \mathcal{N}[\![M]\!](\lambda x.k\ (\mathsf{inl}\,x)) \\ \mathcal{N}[\![\mathsf{inr}\,M]\!]k = \mathcal{N}[\![M]\!](\lambda x.k\ (\mathsf{inr}\,x)) \\ \mathcal{N}\left[\!\!\left[\begin{array}{l}\mathsf{case}\ M\ \{\\ \quad \mathsf{inl}\,x \mapsto N;\\ \quad \mathsf{inr}\,y \mapsto N'\}\end{array}\right]\!\!\right] k = \mathcal{N}[\![M]\!]\left(\begin{array}{ll}\mathsf{case}\ z\ \{ & \\ \lambda z. & \mathsf{inl}\,x \mapsto \mathcal{N}[\![N]\!]k;\\ & \mathsf{inr}\,y \mapsto \mathcal{N}[\![N']\!]k\}\end{array}\right) \\ \mathcal{N}[\![\mathsf{absurd}\ M]\!]k = \mathcal{N}[\![M]\!](\lambda z.\mathsf{absurd}\ z) \\ \mathcal{N}[\![\langle M,N\rangle]\!]k = \mathcal{N}[\![M]\!](\lambda x.\mathcal{N}[\![N]\!](\lambda y.k\ \langle x,y\rangle)) \\ \mathcal{N}[\![\mathsf{fst}\,M]\!]k = \mathcal{N}[\![M]\!](\lambda x.k\ (\mathsf{fst}\,x)) \\ \mathcal{N}[\![\mathsf{snd}\,M]\!]k = \mathcal{N}[\![M]\!](\lambda x.k\ (\mathsf{snd}\,x)) \\ \mathcal{N}[\![\langle\rangle]\!]k = k\ \langle\rangle \\ \mathcal{N}[\![\mathsf{in}\,M]\!]k = \mathcal{N}[\![M]\!](\lambda x.k\ (\mathsf{in}\,x)) \\ \mathcal{N}[\![(\!|M|\!)]\!]k = \mathcal{N}[\![M]\!](\lambda x.k\ (\!|x|\!)) \\ \mathcal{N}[\![\mathsf{out}\,M]\!]k = \mathcal{N}[\![M]\!](\lambda x.k\ (\mathsf{out}\,x)) \\ \mathcal{N}[\![[\![M]\!]]\!]k = \mathcal{N}[\![M]\!](\lambda x.k\ ([\![x]\!])) \end{array}$$

Figure 10: Naive CPS Translation for Core Terms

As well as providing a means to prove termination, the CPS transformation is interesting in its own right as it provides insights into the restricted nature of the concurrency provided by μGV. Furthermore, by composing the CPS translation with the translation from μCP to μGV (§6.3), we obtain a translation from μCP into a typed lambda calculus. This shows that the concurrency of μCP is equivalent to that provided by full reduction in the λ-calculus.

We can make the translation more uniform by factoring it through a *polarization* phase. In particular, polarization allows us to give a single translation for each of the send, fork, and link cases. Polarization provides a way of encoding output session types as input session types and vice-versa. It also leads to a clean way of handling polymorphic session types by uniformly choosing either a positive (output) or a negative (input) representation for session type variables.

6. The μCP Language

In this section, we present the syntax (§6.1) and semantics (§6.2) of μCP, an extension of Wadler's CP calculus with recursive and corecursive types following Baelde's approach to recursion and corecursion in classical linear logic [3]. We then argue that μCP and μGV are equally expressive via semantics-preserving translations from μCP into μGV and vice versa.

Types

$$\begin{array}{ll} \mathcal{K}[\![\mathsf{end}_!]\!] = R & \mathcal{K}[\![!T.S]\!] = \mathcal{K}[\![T]\!] \multimap \mathcal{K}[\![\overline{S}]\!] \multimap R \\ \mathcal{K}[\![\mathsf{end}_?]\!] = R \multimap R & \mathcal{K}[\![?T.S]\!] = (\mathcal{K}[\![T]\!] \multimap \mathcal{K}[\![\overline{S}]\!] \multimap R) \multimap R \end{array}$$

Constants

$$\begin{array}{c} \mathcal{K}[\![\mathsf{send}_!]\!]p\ k = \mathsf{let}\ (x,c) = p\ \mathsf{in}\ (c\ x)\ k \\ \mathcal{K}[\![\mathsf{send}_?]\!]p\ k = \mathsf{let}\ (x,c) = p\ \mathsf{in}\ k\ (c\ x) \\ \mathcal{K}[\![\mathsf{receive}]\!]c\ k = c\ (\lambda x\ c.k\ (x,c)) \\ \mathcal{K}[\![\mathsf{fork}_!]\!]f\ k = k\ (\lambda x.f\ x\ id) \\ \mathcal{K}[\![\mathsf{fork}_?]\!]f\ k = (\lambda x.f\ x\ id)\ k \\ \mathcal{K}[\![\mathsf{wait}]\!]c\ k = c\ (k\ ()) \\ \mathcal{K}[\![\mathsf{link}_!]\!]p\ k = \mathsf{let}\ (c,d) = p\ \mathsf{in}\ k\ (c\ d) \\ \mathcal{K}[\![\mathsf{link}_?]\!]p\ k = \mathsf{let}\ (c,d) = p\ \mathsf{in}\ k\ (d\ c) \end{array}$$

Shallow Polarization

$$S_! ::= !T.S \mid \mathsf{end}_! \qquad S_? ::= ?T.S \mid \mathsf{end}_?$$

$$\begin{array}{ll} \mathsf{send}_! : T \otimes !T.S_! \multimap S_! & \mathsf{send}_? : T \otimes !T.S_? \multimap S_? \\ \mathsf{fork}_! : (S_! \multimap \mathsf{end}_!) \multimap \overline{S_!} & \mathsf{fork}_? : (S_? \multimap \mathsf{end}_!) \multimap \overline{S_?} \\ \mathsf{link}_! : S_! \otimes \overline{S_!} \multimap \mathsf{end}_! & \mathsf{link}_? : S_? \otimes \overline{S_?} \multimap \mathsf{end}_! \end{array}$$

Configurations

$$\begin{array}{rcl@{\qquad}rcl} \mathcal{K}[\![\bullet\, M]\!]k & = & \mathcal{K}[\![M]\!]k & \mathcal{K}[\![(\mathsf{new}\,x)C]\!]k & = & \mathcal{K}[\![C]\!]k \\ \mathcal{K}[\![\circ\, M]\!]k & = & \mathcal{K}[\![M]\!]id & \mathcal{K}[\![z = x \leftrightarrow y]\!]k & = & z\,(x\,y) \\ \mathcal{K}[\![C \parallel_x C']\!]k & = & (\mathcal{K}[\![C]\!]k)\{(\lambda x.\mathcal{K}[\![C']\!]k)/x\} & & & \end{array}$$

Figure 11: CPS Translation for Concurrent μGV and Contexts.

Evaluation Contexts

$$\mathcal{E}[\![E]\!]k = \lambda x.\mathcal{K}[\![E[e]]\!]k$$

Configuration Contexts

$$\begin{array}{c} \mathcal{D}[\![[\,]]\!]f\ k = f\ k \\ \mathcal{D}[\![(\mathsf{new}\,x)D]\!]f\ k = \mathcal{K}[\![D]\!]f\ k \\ \mathcal{D}[\![C \parallel_x D]\!]f\ k = (\mathcal{K}[\![C]\!]k)\{\lambda x.\mathcal{D}[\![D]\!]f\ k/x\} \\ \mathcal{D}[\![D \parallel_x C]\!]f\ k = (\mathcal{D}[\![D]\!]f\ k)\{\lambda x.\mathcal{K}[\![C]\!]k/x\} \end{array}$$

Figure 12: CPS Translation of μGV Contexts.

6.1 Syntax

Figure 13 gives the terms and typing of μCP, an extension of Wadler's process calculus CP with recursive and corecursive types. The syntax of types is that of the propositions of linear logic, extended with least (μF) and greatest (νF) fixed points. As in μGV, we have omitted polymorphism and the exponential modality; their reintroduction is entirely orthogonal to our development. The definition of duality includes the duality of least and greatest fixed points; the dual of an operator is defined by $F^{\perp}(X) = (F(X^{\perp}))^{\perp}$, as for μGV. The terms of μCP are restricted compared to π-calculus in several ways. Most significantly, composition and name restriction are combined in a single syntactic form, and the composed processes are limited to share only the newly introduced name. The forwarding construct $x \leftrightarrow y$ corresponds to the axiom rule in linear logic; it is necessary for the treatment of recursion and for the extension of μCP to include polymorphism.

A Simpler Send. The μCP rule for output is appealing because it corresponds exactly to the linear logic proof rule for $\otimes$. Its correspondence to π-calculus is less direct: the term for output includes name restriction (introducing new name y), output of y along x, and finally a restricted composition (of P and Q). This

Syntax

$$
\begin{array}{lll}
\text{Types} & A, B ::= & A \otimes B \mid A \parr B \mid A \oplus B \mid A \,\&\, B \\
& & \mid \mathbf{1} \mid \bot \mid \top \mid \mathbf{0} \mid X \mid X^\bot \mid \mu F \mid \nu F \\
\text{Operators} & F, G ::= & X.A \\
\text{Labels} & \ell & \in \mathsf{inl}, \mathsf{inr} \\
\text{Processes} & P, Q, R ::= & x[y].(P \mid Q) \mid x(y).P \mid x[].0 \mid x().P \\
& & \mid x[\ell].P \mid \mathsf{case}\ x\ \{P; Q\} \mid \mathsf{case}\ x\ \{\} \\
& & \mid x \leftrightarrow y \mid \mathsf{new}\ x\ (P \mid Q) \\
& & \mid \mathsf{rec}\ x.P \mid \mathsf{corec}\ x[y](P \mid Q)
\end{array}
$$

Typing

$$\frac{}{x \leftrightarrow y \vdash x : A, y : A^\bot} \qquad \frac{P \vdash \Psi, x : A \quad Q \vdash \Psi', x : A^\bot}{\mathsf{new}\ x\ (P \mid Q) \vdash \Psi, \Psi'}$$

$$\frac{P \vdash \Psi, y : A \quad Q \vdash \Psi', x : B}{x[y].(P|Q) \vdash \Psi, \Delta', x : A \otimes B} \qquad \frac{}{x[].0 \vdash x : \mathbf{1}}$$

$$\frac{P \vdash \Psi, x : B, y : A}{x(y).P \vdash \Psi, x : A \parr B} \qquad \frac{P \vdash \Psi}{x().P \vdash \Psi, x : \bot}$$

$$\frac{P \vdash \Psi, x : A}{x[\mathrm{inl}].P \vdash \Psi, x : A \oplus B} \qquad \frac{P \vdash \Psi, x : A \quad Q \vdash \Psi, x : B}{\mathsf{case}\ x\ \{P; Q\} \vdash \Psi, x : A \,\&\, B}$$

$$\frac{}{\mathsf{case}\ x\ \{\} \vdash \Psi, x : \top} \qquad \frac{P \vdash \Psi, x : F(\mu F)}{\mathsf{rec}\ x.P \vdash \Psi, x : \mu F}$$

$$\frac{P \vdash \Psi, y : A \quad Q \vdash y : A^\bot, x : F(A)}{\mathsf{corec}\ x[y](P \mid Q) \vdash \Psi, x : \nu F}$$

Duality

$$
\begin{array}{lll}
(A \otimes B)^\bot = A^\bot \parr B^\bot & \mathbf{1}^\bot = \bot & \bot^\bot = \mathbf{1} \\
(A \parr B)^\bot = A^\bot \otimes B^\bot & \mathbf{0}^\bot = \top & \top^\bot = \mathbf{0} \\
(A \oplus B)^\bot = A^\bot \,\&\, B^\bot & (\mu F)^\bot = \nu(F^\bot) & (X^\bot)^\bot = X \\
(A \,\&\, B)^\bot = A^\bot \oplus B^\bot & (\nu F)^\bot = \mu(F^\bot) & F^\bot(X) = (F(X^\bot))^\bot
\end{array}
$$

Figure 13: μCP Typing Rules

complicates the reduction relation for μCP (which must account for all three behaviors) and the correspondence to μGV (where the rule for send is simpler).

An alternative presentation (following Boreale [12]) avoids the name restriction and composition, as follows:

$$\frac{P \vdash \Psi, x; B, y : A}{x\langle y\rangle.P \vdash \Psi, x : A \otimes B, y : A^\bot}$$

While no longer identical to the $\otimes$ rule, this is closer to the formulation of output in π-calculus. As in our prior work [26], we write $x\langle y\rangle.P$ as syntactic sugar for $x[z].(y \leftrightarrow z \mid P)$.

Recursion and Corecursion. In μCP, recursion and corecursion follow Baelde's extension of classical linear logic to include induction and coinduction [3]. We begin by considering sequent calculus presentations of introduction and elimination rules for induction and coinduction, as follows:

$$\frac{\Psi \vdash F(\mu F)}{\Psi \vdash \mu F} \qquad \frac{F(A) \vdash A}{\mu F \vdash A} \qquad \frac{\Psi, \nu F \vdash A}{\Psi, F(\nu F) \vdash A} \qquad \frac{A \vdash F(A)}{A \vdash \nu F}$$

Note that the hypotheses of the right rule for ν and left rule for μ are restricted to account for linearity. Baelde observes that, when using duality to convert these two-sided sequents to one-sided sequents, the left rule for μ and the right rule for ν collapse, and similarly the right rule for μ and the left rule for ν. This leaves us with only two rules, with term assignments as follows:

$$\frac{P \vdash y^\bot : A, x : F(A)}{\mathsf{corec}\ x(y).P \vdash y : A^\bot, x : \nu F} \qquad \frac{P \vdash \Psi, x : F(\mu F)}{\mathsf{rec}\ x.P \vdash \Psi : x : \mu F}$$

However, there is a problem with this formulation. Suppose that we have some term $Q \vdash A$. We then have the composition $\mathsf{new}\ y\ (Q \mid \mathsf{corec}\, x(y).P) \vdash x : \nu F$. However, we have no hope of reducing this cut, as we have no rule which can prove νF in isolation. We can address this problem by suspending the cut in question, moving it into the ν rule and giving the rule in Figure 13. In our prior work [26] we observe a similar pattern in comparing the $\otimes$ rule to the typical process calculus rule for output. As in that case, the version without the suspended cut may expose reductions not present in the suspended version. Nevertheless, we can still define the simpler term as syntactic sugar:

$$\mathsf{corec}\ x(y).P = \mathsf{corec}\ x[y](y \leftrightarrow z \mid P)$$

Examples. We return to the example of natural numbers to give some flavor of the use of recursion and corecursion in μCP. We can define the type of natural numbers much as before

$$N(X) = \mathbf{1} \oplus X \qquad\qquad Nat = \mu N$$

and we can give very similar definitions of the constructors

$$\begin{aligned} zero_x &= \mathsf{rec}\ x.x[\mathrm{inl}].x[].0 \\ succ_{xy} &= \mathsf{rec}\ x.x[\mathrm{inr}].x \leftrightarrow y \end{aligned}$$

with the expected typings $zero_x \vdash x : Nat$ and $succ_{xy} \vdash x : Nat, y : Nat^\bot$. We can define the addition operation as follows:

$$
\begin{aligned}
plus_{xyz} = \mathsf{corec}\ z[w].(&w\langle x\rangle.w \leftrightarrow y; \\
&w(x).\mathsf{case}\ z\ \{z().w \leftrightarrow x; \\
&\qquad \mathsf{rec}\ x.x[\mathrm{inr}].z\langle x\rangle.z \leftrightarrow w\})
\end{aligned}
$$

where the recursive body of the corec has type $w : Nat \parr Nat^\bot, z : N(Nat^\bot \otimes Nat)$ and so the term has typing $plus_{xyz} \vdash x : Nat, y : Nat^\bot, z : Nat^\bot$. Writing $\underline{n}_z$ to denote the encoding of the natural number n along channel z, we have that

$$\mathsf{new}\ z\ (\underline{2}_z \mid \mathsf{new}\ y\ (\underline{2}_y \mid plus_{xyz}))$$

will reduce to $\underline{4}_x$

6.2 Semantics

The semantics of μCP are given by the cut reduction rules in classical linear logic, extended to account for recursion and corecursion, as shown in Figure 14. We write $fv(P)$ for the free names of process P. Terms are identified up to congruence $\equiv$. Many of the principle cut reductions ($\longrightarrow_{\mathrm{C}}$) correspond to process calculus reductions. The reduction of input against output is complicated by the implicit name restriction and composition inherent in the term structure for output. The new rule for μCP is for rec against corec, and amounts to one unfolding of the corec term. In defining the unfolding, we rely on functoriality for the operators; if $P \vdash x : A^\bot, y : B$, then $\mathsf{map}^F_{x,y}(P) \vdash x : F^\bot(A^\bot), y : F(B)$. (We show functoriality for the positive combinators; the remaining cases can be obtained by switching the channels in the given cases.) We write $\longrightarrow$ for $\longrightarrow^\star_{\mathrm{C}} \longrightarrow^\star_{\mathrm{CC}}$. The following theorem is due to Baelde [3]:

Theorem 12 (Cut elimination)**.** *If $P \vdash \Psi$, then there is some P' such that $P \longrightarrow P'$ and P' is not of the form $\mathsf{new}\ x\ (Q \mid Q')$ for any x, Q, Q'.*

This result corresponds to the termination and deadlock freedom results for μGV: any well-typed process reduces to one that is blocked on external communication. The commuting conversions ($\longrightarrow_{\mathrm{CC}}$) do not correspond to computational steps (and thus, do not

Structural Congruence

$$x \leftrightarrow y \equiv y \leftrightarrow x \qquad \mathsf{new}\, y\, (P \mid \mathsf{new}\, x\, (Q \mid R)) \equiv \mathsf{new}\, x\, (\mathsf{new}\, y\, (P \mid Q) \mid R) \quad \text{if } y \notin fv(R)$$
$$\mathsf{new}\, x\, (P \mid Q) \equiv \mathsf{new}\, x\, (Q \mid P) \qquad \mathsf{new}\, x\, (P_1 \mid Q) \equiv \mathsf{new}\, x\, (P_2 \mid Q) \quad \text{if } P_1 \equiv P_2$$

Functoriality (positive cases)

$$\mathsf{map}^{X.A}_{x,y}(P) = x \leftrightarrow y, \quad X \notin FTV(A) \qquad \mathsf{map}^{F \oplus G}_{x,y}(P) = \mathsf{case}\, x\, \{y[\mathrm{inl}].\mathsf{map}^{F}_{x,y}(P); y[\mathrm{inr}].\mathsf{map}^{G}_{x,y}(P)\}$$
$$\mathsf{map}^{X.X}_{x,y}(P) = P \qquad \mathsf{map}^{X.\mu F}_{x,y}(P) = \mathsf{corec}^{F^{\perp}}\, x(y).\mathsf{rec}\, y.\mathsf{map}^{X.F(\mu F)}_{x,y}(P)$$
$$\mathsf{map}^{F \otimes G}_{x,y}(P) = x(x').y[y'].(\mathsf{map}^{F}_{x',y'}(P\{x'/x, y'/y\}) \mid \mathsf{map}^{G}_{x,y}(P))$$

Primary Cut Reductions

$$\mathsf{new}\, x\, (x \leftrightarrow y \mid P) \longrightarrow_{\mathrm{C}} P\{y/x\}$$
$$\mathsf{new}\, x\, (x[y].(P|Q) \mid x(y).R) \longrightarrow_{\mathrm{C}} \mathsf{new}\, x\, (Q \mid \mathsf{new}\, y\, (P \mid R))$$
$$\mathsf{new}\, x\, (p[\mathrm{inl}].P \mid \mathsf{case}\, x\, \{Q; R\}) \longrightarrow_{\mathrm{C}} \mathsf{new}\, x\, (P \mid Q)$$
$$\mathsf{new}\, x\, (\mathsf{corec}^{F} x[y](P \mid Q) \mid \mathsf{rec}\, x.R) \longrightarrow_{\mathrm{C}} \mathsf{new}\, y\, (P \mid \mathsf{new}\, z\, (Q\{z/x\} \mid \mathsf{new}\, x\, (\mathsf{map}^{F}_{x,z}(\mathsf{corec}^{F} x[y](z \leftrightarrow y \mid Q)) \mid R)))$$

Commuting Conversions

$$\mathsf{new}\, z\, (x[y].(P \mid Q) \mid R) \longrightarrow_{\mathrm{CC}} x[y].(\mathsf{new}\, z\, (P \mid R) \mid Q) \qquad \mathsf{new}\, z\, (\mathsf{case}\, x\, \{P; Q\} \mid R) \longrightarrow_{\mathrm{CC}} \mathsf{case}\, x\, \{\mathsf{new}\, z\, (P \mid R); \mathsf{new}\, z\, (Q \mid R)\}$$
$$\mathsf{new}\, z\, (x[y].(P \mid Q) \mid R) \longrightarrow_{\mathrm{CC}} x[y].(P \mid \mathsf{new}\, z\, (Q \mid R)) \qquad \mathsf{new}\, z\, (\mathsf{case}\, x\, \{\} \mid P) \longrightarrow_{\mathrm{CC}} \mathsf{case}\, x\, \{\}$$
$$\mathsf{new}\, z\, (x(y).P \mid Q) \longrightarrow_{\mathrm{CC}} x(y).\mathsf{new}\, z\, (P \mid Q) \qquad \mathsf{new}\, z\, (\mathsf{rec}\, x.P \mid Q) \longrightarrow_{\mathrm{CC}} \mathsf{rec}\, x.\mathsf{new}\, z\, (P \mid Q)$$
$$\mathsf{new}\, z\, (x().P \mid Q) \longrightarrow_{\mathrm{CC}} x().\mathsf{new}\, z\, (P \mid Q) \qquad \mathsf{new}\, z\, (\mathsf{corec}\, x[y](P \mid Q) \mid R) \longrightarrow_{\mathrm{CC}} \mathsf{corec}\, x[y](\mathsf{new}\, z\, (P \mid R) \mid Q)$$
$$\mathsf{new}\, z\, (x[\mathrm{inl}].P \mid Q) \longrightarrow_{\mathrm{CC}} x[\mathrm{inl}].\mathsf{new}\, z\, (P \mid Q)$$

Figure 14: μCP Reduction Rules

correspond to reductions in process calculi), but play a crucial role in cut elimination by moving remaining internal communication behind any external communication.

6.3 Translations between μCP and μGV

We conclude our discussion of μCP by discussing its relationship with μGV. Our previous work [26] considers a functional calculus GV and a process calculus CP, similar to μGV and μCP but lacking recursion and corecursion. In that setting we give translations $\mathcal{C}[\![-]\!]$ and $\mathcal{G}[\![-]\!]$ from GV configurations to CP terms and vice versa and show that these translations preserve both typing and semantics.

We have devised similar translations between μGV and μCP. We omit the details of the translations here. This is not only for reasons of space; the details of the translations are identical but for the treatment of recursion, and the recursive forms in μGV and μCP are already quite similar. As in the case of the non-recursive calculi, these translations preserve both typing and semantics.

Theorem 13. *If $P \vdash \Psi$, then $\mathcal{G}[\![\Psi]\!] \vdash^{\circ} \mathcal{G}[\![P]\!] : \mathsf{end}_!$.*

Theorem 14. *If $P \vdash \Psi$ and $P \longrightarrow_C Q$ then $\mathcal{G}_C[\![P]\!] \longrightarrow^{+} \approx_{-} \mathcal{G}_C[\![Q]\!]$.*

Theorem 15. *If $\Gamma \vdash C : T$, then $\mathcal{C}[\![C]\!]z \vdash \mathcal{C}[\![\Gamma]\!], z : \mathcal{C}[\![T]\!]^{\perp}$.*

We write $\Longrightarrow$ for $(\equiv \longrightarrow \equiv)^{+}$.

Theorem 16. *If $\Gamma \vdash C : T$ and $C \longrightarrow C'$, then $\mathcal{C}[\![C]\!]z \Longrightarrow \mathcal{C}[\![C']\!]z$.*

The simulation of μGV evaluation by μCP evaluation is straightforward. The simulation of μCP evaluation by μGV evaluation relies on the same administrative weak bisimulation results needed to show that the core μGV calculus simulates its extension with recursive session types, but introduces no other difficulties.

7. Related Work

Session Types and Linear Logic. Session types were originally introduced by Honda [22] as a typing discipline for a CCS-like process calculus. Takeuchi et al [30] and Honda et al. [23] extended the original approach to include delegation and recursion. Honda's system relied on a substructural type system, and borrowed some syntax from linear logic, but did not draw a direct connection between the systems nor suggest the connection between the input and output session types and the $\otimes$ and $\parr$ connectives. Abramsky [1] and Bellin and Scott [5] give interpretations of linear logic proofs as π-calculus processes, and of cut elimination as π-calculus reduction. Their interpretations of $\otimes$ and $\parr$ are very different from the interpretations of input and output in session types. Caires and Pfenning [13] give the first formal correspondence between session types and linear logic, interpreting the propositions of intuitionistic linear logic as session types, and showing that π-calculus reduction corresponds with cut reduction. As a consequence of the latter correspondence, they show that cut elimination in linear logic proves deadlock freedom for session-typed π calculus terms. Vasconcelos et al. [33] and Gay and Vasconcelos [20] consider functional languages extended with session-typed concurrency. The functional fragments of their calculi are generally less fully featured than ours (for example, they omit sums) while their concurrent fragments include non-determinism and deadlock. Wadler [34] presents a process calculus, called CP, similar to that of Caires and Pfenning, but based on classical rather than intuitionistic linear logic. He also gives a functional calculus, called GV, and similar to that of Gay and Vasconcelos. He shows a type-preserving translation from GV to CP; however, his GV is less expressive than CP. Our prior work [26] introduces a more expressive variant of GV, based on Wadler's, and gives both a direct semantics and semantics-preserving translations to and from Wadler's CP.

Recursive and Corecursive Definition. The interpretation of recursive data types, and their connection to recursive functions, has been studied extensively; we highlight the direct precursors of our approach. Goguen et al [21] proposed initial algebras and their corresponding folds as a means for understanding recursive data types and their use. Meijer et al. [29] characterized the use of both folds and unfolds, among other patterns, in the definition of recursive functional programs. The coincidence of least and greatest fixed points for data type constructors in many models was first observed by Freyd [19]; he argues that this observation justifies the use of such fixed points for recursive data types. Baelde and Miller [4]

first described an extension of linear logic with induction and coinduction, encoded using the exponential modality and second-order quantification. Baelde [3] treats induction and coinduction without encoding; in particular, he gives a cut reduction rule for recursive and corecursive terms, and shows cut elimination directly.

Recursive Session Types. There have been several recent developments of recursive session types and their relationship with linear logic. We highlight three closely related to our development.

Toninho et al [32] present a system with recursive session types based on intuitionistic linear logic extended with corecursion. They arrive at a similar (albeit intuitionistic) typing discipline for corecursive session types to ours (§2.3), and give a direct proof of termination for the resulting system (without encoding). However, their approach differs from ours in several significant ways. First, they treat recursive processes as primitive, and so do not expose the connection with recursive data types. In contrast, we believe that the parallels with data types (and thus, our ability to present a simple core calculus) is one of the principal benefits of our approach. One consequence is that they have only corecursive processes ($\nu^!, \nu^?$ in our notation), but not recursive processes ($\mu^?, \mu^!$) nor the possibility of identifying greatest and least fixed points. Finally, our session types are classical, while theirs are intuitionistic. One consequence of our approach is that we are explicit about the role of duality, and thus identify a new notion of duality for recursive session types, while their notion of duality is implicit in the type system. We see the similarities, despite theoretical and methodological differences, as indicative of the strength of both approaches.

Dardha [16] gives an encoding of recursive session-typed π-calculus into recursive (non-linear) π-calculus, and shows that this encoding preserves both typing and semantics. Her encoding is based on self-referential replicated processes, and thus supports arbitrary non-termination, while not attempting to guarantee deadlock or livelock freedom. She adopts a coinductive definition of duality from Bernardi et al. [8], which relies on partially unfolding recursive types at each computation of their duals.

Bono and Padovani [10] and Bernardi and Hennessy [6] independently observed that the standard definition of duality for recursive session types fails when recursion occurs in a carried type. Bono et al [11] present a session-typed functional language with self-dual recursion. They do not attempt to enforce termination or distinguish recursion and corecursion, but do present a definition of duality similar to the one that we propose for self-dual recursive session types (§3.2). Bernardi et al [8] systematically study several duality relations, and propose a notion of session typing independent of the particular duality relation. They also give a coinductive characterization of duality, and suggest a syntactic instance of their characterization. A particular concern of their work, absent from ours, is subtyping: a process may offer more choices than those from which its partner selects. However, their definitions are more complex than ours even without considering subtyping; in particular, they rely on partially unfolding recursive types in each computation of their duals.

8. Future Work

We have presented a core concurrent linear λ-calculus with recursive and corecursive data types, and shown how to encode recursive and corecursive session types and processes in this calculus. We have shown that our type system guarantees termination and lock freedom, giving modular proofs which can easily be extended to encompass additional features. We have given a natural, semantically justified approach to extending our system to non-terminating (but still productive) computation. Finally, we have related our calculus to a process calculus based on classical linear logic with induction and coinduction, giving strong logical foundations to our work.

The model of concurrency in μGV (like much of the work on logically founded session types) is somewhat limited. By ruling out deadlock, livelock, and in particular data races, it also rules out interesting forms of concurrency. For instance, there is no way of modeling a non-deterministic stateful service such as an online book store in which it should be possible for one customer to observe that a different customer bought the last copy of a book. This raises the question: can the logically founded approach be extended to encompass more realistic forms of concurrency? Recently, Atkey et al. [2] demonstrated that conflating dual propositions in a process calculus based on classical linear logic captures non-determinism, shared state, and more expressive concurrency patterns. On the one hand we would like to transfer these results to GV. On the other hand we would we would like to transfer ideas for the current paper to extend the work of Atkey et al. to incorporate conflated fixed points.

While the development of μGV in this paper is largely theoretical, we believe it can also inform practical implementations of session types, including our implementation of session types for the Links web programming language [31]. Indeed, the Links implementation of session types is based on a core calculus FST (System F with Session Types), a polymorphic variant of GV [27]. We would like to investigate the relationship between μGV with FST, while also taking into account practical considerations.

In this paper we have focused on binary session types. Carbone et al. [14] give a logical account of multiparty session types based on CP. It would be interesting to adapt this work to the μGV setting by following our general approach to relating variants of CP and GV.

Acknowledgments. Thanks to Giovanni Bernardi, Ornela Dardha, Simon Fowler, Philip Wadler, and the anonymous referees for helpful feedback. This work was funded by EPSRC grant number EP/K034413/1.

References

[1] S. Abramsky. Proofs as processes. *Theor. Comput. Sci.*, 135(1):5–9, 1994.

[2] R. Atkey, S. Lindley, and J. G. Morris. Conflation confers concurrency. In S. Lindley, C. McBride, P. W. Trinder, and D. Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 32–55. Springer, 2016.

[3] D. Baelde. Least and greatest fixed points in linear logic. *ACM Trans. Comput. Logic*, 13(1):2:1–2:44, Jan. 2012.

[4] D. Baelde and D. Miller. Least and greatest fixed points in linear logic. In N. Dershowitz and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007, Proceedings*, volume 4790 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2007.

[5] G. Bellin and P. J. Scott. On the π-Calculus and linear logic. *Theoretical Computer Science*, 135(1):11–65, 1994.

[6] G. Bernardi and M. Hennessy. Using higher-order contracts to model session types (extended abstract). In P. Baldan and D. Gorla, editors, *CONCUR 2014*, volume 8704 of *Lecture Notes in Computer Science*, pages 387–401. Springer, 2014.

[7] G. Bernardi and M. Hennessy. Using higher-order contracts to model session types. *CoRR*, abs/1310.6176v4, 2015.

[8] G. Bernardi, O. Dardha, S. J. Gay, and D. Kouzapas. On duality relations for session types. In *Trustworthy Global Computing - 9th International Symposium, TGC 2014, Rome, Italy, September 5-6, 2014. Revised Selected Papers*, pages 51–66, 2014.

[9] R. S. Bird and O. de Moor. *Algebra of programming*. Prentice Hall International series in computer science. Prentice Hall, 1997.

[10] V. Bono and L. Padovani. Typing copyless message passing. *Logical Methods in Computer Science*, 8(1), 2012.

[11] V. Bono, L. Padovani, and A. Tosatto. Polymorphic types for leak detection in a session-oriented functional language. In D. Beyer and M. Boreale, editors, *Formal Techniques for Distributed Systems - Joint IFIP WG 6.1 International Conference, FMOODS/FORTE 2013, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013, Florence, Italy, June 3-5, 2013. Proceedings*, volume 7892 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2013.

[12] M. Boreale. On the expressiveness of internal mobility in name-passing calculi. In U. Montanari and V. Sassone, editors, *CONCUR '96, Concurrency Theory, 7th International Conference, Pisa, Italy, August 26-29, 1996, Proceedings*, volume 1119 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 1996.

[13] L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In P. Gastin and F. Laroussinie, editors, *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, volume 6269 of *Lecture Notes in Computer Science*, pages 222–236. Springer, 2010.

[14] M. Carbone, S. Lindley, F. Montesi, C. Shürmann, and P. Wadler. Coherence generalises duality: a logical explanation of multiparty session types. In *CONCUR*. LIPICS, 2016. To appear.

[15] O. Danvy and L. R. Nielsen. A first-order one-pass CPS transformation. In M. Nielsen and U. Engberg, editors, *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings*, volume 2303 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2002.

[16] O. Dardha. Recursive session types revisited. In *Proceedings Third Workshop on Behavioural Types, BEAT 2014, Rome, Italy, 1st September 2014.*, pages 27–34, 2014.

[17] O. Dardha, E. Giachino, and D. Sangiorgi. Session types revisited. In D. D. Schreye, G. Janssens, and A. King, editors, *Principles and Practice of Declarative Programming, PPDP'12, Leuven, Belgium - September 19 - 21, 2012*, pages 139–150. ACM, 2012.

[18] A. Filinski. Linear continuations. In R. Sethi, editor, *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, pages 27–38. ACM Press, 1992.

[19] P. Freyd. Algebraically complete categories. In G. R. Aurelio Carboni, Maria Cristina Pedicchio, editor, *Category Theory - Proceedings of the International Conference held in Como, Italy, July 22–28, 1990*. Springer, 1990.

[20] S. J. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(01):19–50, 2010.

[21] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *J. ACM*, 24(1):68–95, 1977.

[22] K. Honda. Types for dyadic interaction. In E. Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993.

[23] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In C. Hankin, editor, *ESOP*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.

[24] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, 1999.

[25] J. Lévy and L. Maranget. Explicit substitutions and programming languages. In *Foundations of Software Technology and Theoretical Computer Science, 1999*, volume 1738 of *LNCS*. Springer, 1999.

[26] S. Lindley and J. G. Morris. A semantics for propositions as sessions. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 560–584, 2015.

[27] S. Lindley and J. G. Morris. Lightweight functional session types, 2015. Draft `http://homepages.inf.ed.ac.uk/slindley/papers/fst-draft-february2015.pdf`.

[28] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 260–267, New York, NY, USA, 1988. ACM.

[29] E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer, 1991.

[30] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In C. Halatsis, D. G. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *PARLE '94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings*, volume 817 of *Lecture Notes in Computer Science*, pages 398–413. Springer, 1994.

[31] The Links Team. Links, 2016. `http://groups.inf.ed.ac.uk/links`.

[32] B. Toninho, L. Caires, and F. Pfenning. Corecursion and non-divergence in session-typed processes. In *Trustworthy Global Computing - 9th International Symposium, TGC 2014, Rome, Italy, September 5-6, 2014. Revised Selected Papers*, pages 159–175, 2014.

[33] V. T. Vasconcelos, S. J. Gay, and A. Ravara. Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.*, 368(1-2):64–87, 2006.

[34] P. Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014.